2009-06-01

Felix - A Simulation-Tool for Neural Networks (and Dynamical Systems). User Guide.

Wennekers, Thomas

Felix - a Simulation-Tool for Neural Networks

(and Dynamical Systems)

USER GUIDE

Thomas Wennekers

Centre for Theoretical and Computational Neuroscience
University of Plymouth
PL4 8AA Plymouth, Devon, United Kingdom

May 15, 2009

Dear valued Reader

This is the User Guide of "Felix", a simulation environment for neural networks and dynamical systems. It is C-based and provides a simple to use graphical interface as well as real time control of simulation parameters. The main aim of the tool is to simplify the implementation and simulation of distributed neural networks consisting of either homogeneous pools or 2-dimensional layers of simple spiking neurons. Other, more general dynamical systems can be implemented and visualised as well, and several examples are provided (coupled map lattice, coupled Roessler oscillators). The simulation of conductance-based neuron types is possible but only marginally supported.

The tool can make use of code-parallelisation on three levels: single CPU vectorisation using BLAS-SSE2, SMP-shared memory parallelism via OpenMP (threads), and the message passing interface (MPI) for computer clusters. Hybrid BLAS/OpenMP/MPI code is possible, e.g., for use on SMP-clusters. Felix can be downloaded from http://www.pion.ac.uk, which provides run-time libraries, the development tool, and a couple of examples. Source code is also available and, beside on Linux single- and multi-processor computers, and Linux Beowulf clusters, can be compiled and run on Windows using the Cygwin-Linux emulator.

Have fun
Thomas Wennekers

Contents

Chapter 1

Introduction

1.1 Overview

This is a preliminary version of a User Guide for "Felix" - A simulation tool for neural networks and dynamical systems. It is currently being written. This introduction, the quick-start guide in section 2, sections 3 about the graphical user interface and 5 about file I/O, the description of the main function libraries in section 4, and the appendix about installation A are something like in a readable state. The examples (section 10) and the section about parallel Felix extensions 9 are still mostly empty or bad. You would probably want to consult the examples that come with Felix directly, if you think about using the tool and want to learn more about how to do so. Serial and parallel example programs are available.

Felix is a development tool for neural network and dynamical systems simulations. It is C-based and provides a simple to use graphical interface as well as a core of routines needed in many applications. Routines required in special applications can easily be added. Felix is best suited for one and two-dimensional network models, but other topologies are possible as well.

1.2 The main philosophy of Felix

Main philosophy of Felix is to consider a neural network or more general dynamical system as a set of variables, x, which obey a certain dynamics, and a second set of parameters, p, which control this dynamics. Canonical examples are difference schemes, $x(t+1) = f(x(t); p)$ or differential equations, $dx/dt = f(x; p)$, which are omni-present in neural network and dynamical systems theory.

The Felix core implements and solves these dynamical equations and the graphical interface then presents the variables in various possible views like graphs and raster plots over time, images or functions displayed per single time-step, or xy-plots as on an oscilloscope.

The parameters of a simulation are further displayed as a collection of buttons and sliders in the graphical user interface, whereby it becomes possible to change them while the simulation is running and immediately observe the induced changes in the system dynamics. Figure 1.1 displays the graphical user interface of a typical small Felix program (actually a network of so-called integrate-and-fire neurons).

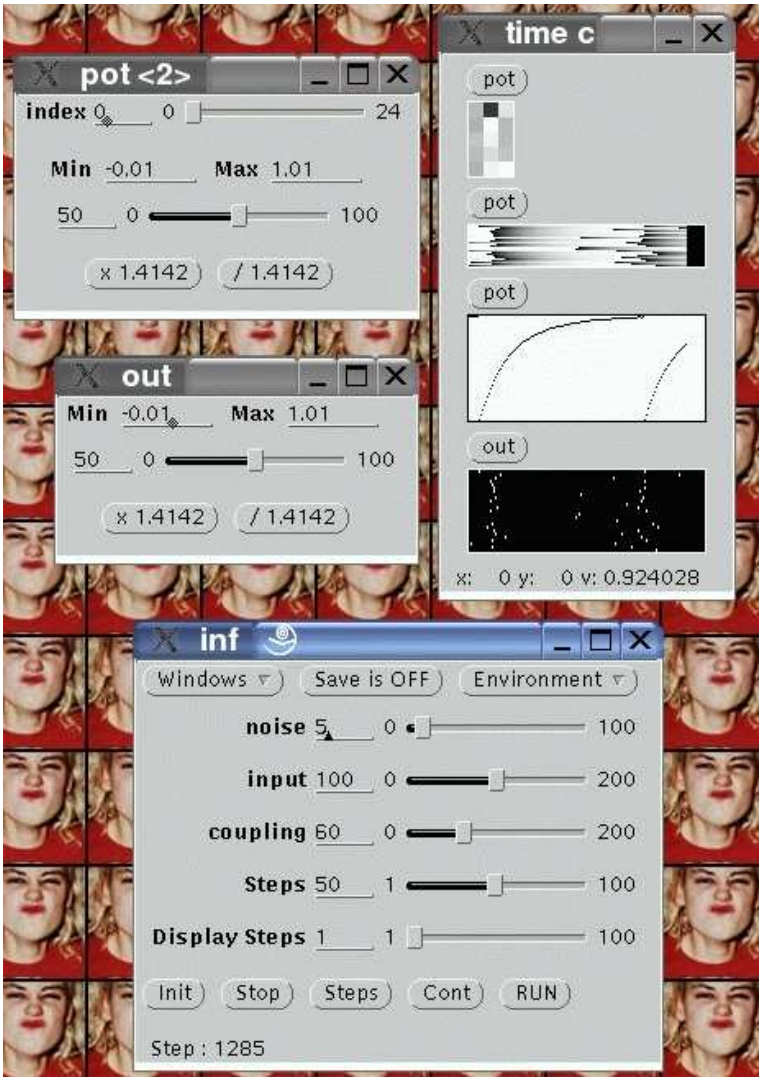

Figure 1.1: A typical Felix simulation showing a panel with control parameters at the bottom and windows for displaying variables of a running simulation at the top. Changing parameters is immediately reflected in the displayed variables.

A second design principle of Felix is that it aims at either "pool" networks comprising (more or less) large ensembles of potentially all-to-all connected units, or at layered one- and two-dimensional networks with a neighbourhood topology. Several such "pools" or "layers" may be combined into larger super-networks, see Figure 1.2. The first type of network model appears if local ensembles of cells in the brain are considered, the second if the focus is on the distributed processing within whole brain areas. In more general dynamical systems the first alternative refers to globally connected systems, whereas the second turns up, e.g., in partial differential equations and integro-differential equations. The core of the Felix simulation tool provides a number of often used routines to implement and simulate neural structures of the respective architecture, i.e., randomly connected pools, associative memories, or distributed systems with Gaussian or DOG (diference of Gaussian) lateral coupling kernels.

Recently Felix has been extended towards supporting various kinds of code parallelisation. Philosophy here is to simplify the development of parallel code for the types of networks described above as much as possible. Using about a handful of simple constructs it is now in fact possible to write

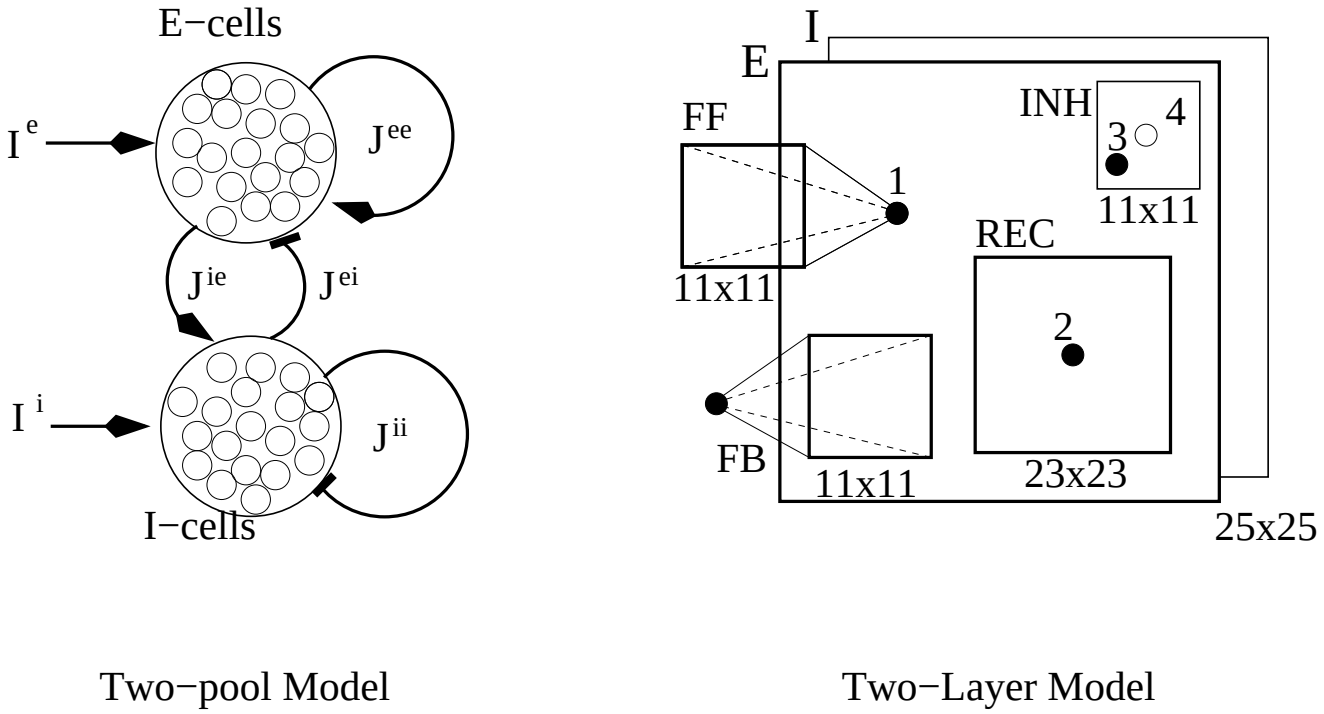

Figure 1.2: Left: A typical pool model (Wilson-Cowan Oscillator). Right: A two-layer, excitatory-inhibitory topographic neural field.

Felix programs that can be compiled on single CPU machines, where they reveal a graphical user interface, but that run also on Beowulf computer clusters. Small programs can therefore run on a PC or laptop, where the GUI and real-time simulation control nicely support an understanding of what is going on in the simulation. The same simulation, however, can now be easily scaled up and run on a much larger scale on a computer cluster without no or only small changes at the source code.

1.3 A little Felix History

Felix is old. The original program was written about 1990/91 in "multiC", a dialect of C for the parallel computer "Wavetracer", which (in the version we had available at that time at the University of Ulm, Germany) consisted of 4096 one-bit processors running at 8MHz in a SIMD-architecture (single instruction multiple data – each processor does the same on possibly different data). Each processor had something like 16MBit local memory and the processor grid was freely configurable as a 1, 2 or 3-D array. The early Felix was meant to serve as a graphical interface for that machine. The Wavetracer was about 20 times faster than a standard Sun-Workstation 15 years ago. When standard workstations became quicker, and in particular quicker than the Wavetracer, I ported Felix to the SunOs and Solaris operating systems, and later, when I discovered that even cheap laptops are faster than standard Sun-workstations, I further ported it to Linux. Now, I am almost exclusively using it under Linux on desktops, laptops, and computer clusters.

Because Felix is old it makes use of an outdated windows toolkit called XView. For a while that was standard for Sun X11 applications with the Open-Look look and feel. However, Sun stopped

developing XView further in about 1995. Meanwhile it has been replaced by more modern toolkits like Motif, QT, and other packages. Although I often thought I should, I never found the time to recode the GUI using a modern toolkit. XView is still available and comes with some Linux distributions. It might however be that it is not installed on your machine by default. I am not sure it is available in 64 bit at the moment. You don't need the graphics libraries if you want to use the tool on computer clusters. Graphical interfaces don't make too much sense in high performance computing.

Some resources:

- Open-Look FAQ: http://www.faqs.org/faqs/open-look/01-general/
- XView FAQ: http://www.faqs.org/faqs/open-look/03-xview/
- O'Reilly provides free books about XView programming on their homepage http://www.oreilly.com/openbook/openlook
- Dr Andreas Knoblauch, a former collegue at the University of Ulm (now at Honda Research, Offenbach, Germany) has written C++ extensions for Felix which you can find here: http://www.informatik.uni-ulm.de/ni/mitarbeiter/AKnoblauch.html

Since relatively recently I am experimenting with parallelised Felix versions. This means Felix gets back to its roots, to parallel computers. The code contained in the distributed Felix version should be considered preliminary and is not well tested. However, it supports hybrid OpenMP/MPI code, which can be very useful for some types of layered network models of the brain. We are studying such models at the University of Plymouth as part of two big research projects: The EU-integrated FACETS project (comprising more than 100 scientists) and the UK-wide COLAMN projects (ca 10 research groups).

1.4 Installation Notes

Throughout this Guide it will be assumed that a functioning serial Felix evironment with graphical user interface is available. Only few sections in addition assume a parallel installation, in particular chapter 9. Appendix A explains, how Felix can be installed on serial and parallel computers, and computer clusters.

Chapter 2

Getting Started

This section presents the main features of Felix by showing a simple example and how it is implemented. The example will consist of a small network of leaky-integrate-and-fire neurons. It will demonstrate how a typical Felix program is structured, how a simulation can be controled by the graphical user interface, and how the simulated data can be conveniently written to files on disc.

2.1 General Program Structure

A Felix application consists of a single C-file. Each application needs to define five subroutines that define the GUI, the output of some data to files, a main-initialisation routine which is called once at start up, an initialisation routine which is called each time a simulation is reset, and a step-routine which contains everything to do in a single simulation step. Some or all of these functions can be empty. The smallest Felix program hence reads:

```
// The most simple Felix program

# include <felix.h>

NO_DISPLAY
NO_OUTPUT
main_init(){}
init(){}
step(){}
```

$< felix.h >$ is the main Felix header file that always has to be included and by itself includes several other header-files neccessary for proper compilation.

The macro `NO_DISPLAY` in the example actually expands to MakeDisplay(){}, e.g., an empty declaration of the graphical user interface. Similarly, the macro `NO_OUTPUT` likewise expands to MakeOutput(){}, an empty declaration of output to files. Simple examples for the GUI and file output follow below. The GUI is treated in detail in chapter 3 and File Output in chapter 5.

The *main_ init()*-routine contains initialisations needed only once during execution of a simulation program. It is executed when the program starts. It may load data from files or settings of

Figure 2.1: Graphical user interface generated by the minimal Felix program given in section 2.1.

parameter-values not accessible by sliders. If the application uses dynamically allocated vectors or arrays, memory for these variables MUST be allocated in main_init(), too, in particular if the variables are supposed to be displayed in the GUI.

The *init()*-routine in contrast is invoked each time a simulation is reset. The GUI provides init- and run-buttons in the main-window to do this. It typically contains code to initialise variables randomly. In conjunction with an additional counter variable in the code that increments each time a reset is performed the init-routine can also be used to scan a parameter range systematically and intialise each simulation in a well defined state using that counter.

The *step()*-function contains all things to be executed in a single simulation step. There is no constraint about the content of this function, but in general it will comprise functions to iterate the dynamics of the simulated systems and possibly also to do some data analysis. The step()-function is repeatedly called if a simulation is in run-mode as long as it is not explicitly stopped. The GUI further supports single- and multi-step modes, in which case the step-routine is executed once or a fixed number of times.

The above trivial Felix program can already be compiled and executed. For that, the code has to be stored in a C-file, i.e., a file called <sim_name>.c, where <sim_name> is some basename (e.g., "empty", because all subroutines are empty functions). Calling "Felix <sim_name>" (i.e., "Felix empty") compiles the program and generates an executable called <sim_name> ("empty"), which can be run from the command line. This should pop up the main-window of the simulation, which should look as displayed in Figure 2.1. (Note: The Felix example directory should contain an "empty.c" function, as well as others.)

The graphical user interface in Figure 2.1 contains simulation control elements that by default appear automatically in the GUI of each simulation program. The top label bar reflects the (base-)name of the compiled program. The "Windows-"button in general comprises a list of defined windows, but in our simple example this list is empty. The "Environment-"button in contrast contains several entries (not shown) that allow to store and load parameter settings for the sliders following below. The "Steps" and "Display-Steps" sliders control the multi-step and display mode of the GUI, respectively. If "Display-Steps" differs from 1, the variable windows (none is shown since they are empty, but see later) are updated only at the respective interval. This is useful to compress time in the display if the simulation step-size is small; it can sometimes also help to speed up simulations, because updating the display needs some time. The "Steps"-slider in the GUI cooperates with the Step-button just below it. If the simulation is in multi-step mode (Display-Steps > 1), the "Steps"-slider specifies how many steps are executed until the simulation stops again, after the "Steps"-button has been pressed. This means, the bottom-row buttons control the overall execution of a simulation: Each time the Init-button is pressed the init()-routine is called. "Run" also calls the init()-routine, but afterwards the step()-routine iteratively – this is

the standard simulation mode. "Stop" stops a simulation, "Step" runs a certain number of steps as explained above, and "Cont" (continue) enters the standard run mode again after a simulation had been stopped. Finally, the footer of the GUI main window contains a counter of the simulation step.

2.2 Example: Leaky-integrate-and-fire Neural Network

We now consider a more interesting example that indeed simulates something. This is a neural network comprising a certain number ($N = 100$) of noisy leaky-integrate-and-fire neurons coupled randomly in a network. These simple neurons are described by membrane potentials x_i that integrate incoming input as low-pass filters with time-constant τ. If a potential crosses a firing threshold of 1 from below it is reset to zero and a spike is emitted. Spikes are represented by a second array of binary variables, z_i, $i = 1, \ldots, N$. Equation (2.1) describes the membrane dynamics and (2.2) the resest at threshold crossings:

$$\tau \frac{dx(t)}{dt} = -x(t) + I + \frac{J_0}{N} \sum_{i=1}^{N} J_{ij} z_j(t) + \sigma \eta_j(t) \tag{2.1}$$

$$\text{if} \quad x_i(t) \geq 1 \quad \text{then} \quad z_i(t) = 1, \text{ and } x_i(t) = 0 \quad \text{else} \quad z_i(t) = 0. \tag{2.2}$$

$\tau (= 10)$ in (2.1) is the membrane time constant and $J_0 = 1.1$ sets the coupling strength between units globally. The J_{ij} in contrast are individual couplings/synapses between pairs of neurons. In the simulation they are independent and identically distributed (i.i.d) Gaussian random numbers with mean 1 and standard deviation 0.4. The $\eta_i(t)$ in (2.1) are furthermore i.i.d. temporally Gaussian white noise processes with mean 0 and standard deviation 1. The factor σ scales this "noise" injected into the individual cells.

Networks of this type have been intensively studied in Neural Network Theory.

The following code implements the network model:

```
/* Example-program: inf.c  */

# include <felix.h>

# define N   100    /* number of neurons      */
# define tau  10.   /* membrane time constant */

float I = 1.1,      /* Common input to units  */
      J0 = 1.1,     /* Coupling strength      */
      sigma = .1;   /* noise level            */

Vector  x;          /* potentials             */
Matrix  J;          /* connections            */
bVector z;          /* vector of spikes       */
Vector  v;          /* auxiliary variable     */

NO_DISPLAY
NO_OUTPUT
```

```
int main_init()
{
  /* init. random number generator and stepsize */
  randomize( time(NULL) );
  SET_STEPSIZE( .1 )

  /* allocate vectors and matrices */
  J = Get_Matrix( N, N );
  x = Get_Vector( N );
  z = Get_bVector( N );
  v = Get_Vector( N );
}

int init()
{
  int i;

  Clear_bVector(N,z);
  Clear_Vector(N,v);

  /* init. potentials with random values between 0 and 1 */
  for (i=0; i<N; i++)
    x[i] = equal_noise();

  /* init. J with gaussian distr. random numbers */
  Make_Matrix( N, N, J, 1.0/N, .4/N  );
}

int step()
{
  int i;

  for (i=0;i<N;i++)  // leaky integration for all neurons
    leaky_integrate ( tau, x[i],
                  I + J0*v[i] + sigma*gauss_noise() );

  Fire_Reset( N, x, 1.0, 0.0, z );  // firing and reset

  bMult( N, N, J, z, v ); // redistribution spikes
}
```

Observe the general structure of the code. First felix.h is included and as (some) parameters of the model are defined as macros (this could also be variables). Then arrays for the neural variables x, J, z and an auxilliary array v are declared. The code still does define an empty GUI and data output routine (NO_DISPLAY and NO_OUTPUT). After that the three obligatory functions main_init(), init(), and step() follow.

main_init() initialises the random number generator and sets the simulation time-step to 0.1. Afterwards the routine allocates the three vectors v, x, z, and the array J. Note that the z-array is a "bVector" – a *binary* Vector. [Many functions in Felix operate either on floating point vectors and matrices or on binary ones, where binary values (0/1) are represented by the C-type "char".]

The init()-funtion initialises the data-arrays: v and z are cleared, ie., set to 0; the potentials are set to equally distributed random numbers in the range [0,1[; and the coupling matrix J is filled with i.i.d. Gaussian random numbers, N(1., 0.4).

Finally, the step()-routine implements the dynamics of the network. It mainly uses functions from the Felix libraries. The leaky integration in equation (2.1) is coded explicitly using the macro "leaky_integrate", which implements a simple Euler-scheme to integrate the low-pass dynamics. "Fire_Reset()" afterwards does the thresholding part of the leaky-integrate-and-fire dynamics, and "bMult()" computes the Matrix-Vector product between the coupling Matrix J and the binary vector of spikes z. The result v is used in the leaky integration in the next step.

Again, the code shown can be compiled and run using Felix, but since it neither defines graphical nor file-output, we would not be able to observe what the network is doing. The interface would just look as in Figure 2.1 with now nice windows or file output at all. Therefore, we next add some graphical output.

2.3 Adding a Graphical User Interface

The graphical user interface serves different tasks, the two most important are displaying variables of the simluation and providing sliders to control it (other task concern file I/O and saving/loading parameters). In the first case the information flow is from the running simulation to the GUI, whereas in the second it is the other way round – the user changes sliders, which in turn modify simulation parameters. The next two sub-sections explain how these tasks are set up.

In general the function MakeDisplay() represents the main-interface between the C-code and the XWindows-System. It contains statements that define how variables shall be displayed on the screen and, if needed, declares buttons (called switches) and sliders, which allow for interactive control of a running simulation. MakeDisplay always generates a main-window with several buttons and sliders, which are used to control the simulator-kernel even if the MakeDisplay() function is explicitly declared empty or the macros NO_DISPLAY is used (which does the same), see Figure 2.1.

2.3.1 Displaying Views on Variables

As outlined in section 1.2 a simulation can be considered a dynamical system comprising variables and parameters. Variables are displayed to the user and parameters can be used to modify the simulation online. The code below shows how a typical Graphical User Interface for the leaky-integrate-and-fire neural network program can be declared.

```
BEGIN_DISPLAY

  WINDOW("time courses")

    IMAGE( "x", AR, AC, x, VECTOR,  10, 10, 0.0, 1.0, 4)
    RASTER( "x", NR, AC, x, VECTOR,  N, 0, 0.0, 1.0, 1)
    GRAPH( "x", NR, AC, x, VECTOR,  N, 0, 0, 0, -.01, 1.01 )
    RASTER( "z", NR, AC, z, bVECTOR, N, 0, -.01, 1.01, 2)
```

```
  WINDOW("couplings")

    IMAGE( "J", AR, AC, J, CONSTANT MATRIX, N, N, -4./N, 4./N, 4)

END_DISPLAY
```

The macros BEGIN_DISPLAY and END_DISPLAY enclose the definition of a GUI; they expand into a MakeDisplay(){} function body (thus, you can also define this function directly without using the macros). Everything between the BEGIN_ and END_DISPLAY macros is executed when the GUI is build. In the present example two windows are defined with names "time courses" and "couplings", respectively. It is possible to define an arbitrary number of such display windows.

Each display window can in turn comprise an arbitrary number of so-called "views". A view is a view of a variable, e.g., a scalar, vector, or a matrix. Each variable can be viewed in different ways, and section 3 describes the possibilities in detail. Here, it may suffice to observe that the window "couplings" displays the $N \times N$ coupling matrix J as an IMAGE, i.e., a grey-scale coded picture that reflects the values of the matrix entries. Because J is declared as a CONSTANT MATRIX in the IMAGE-definition, the image of J is updated only once, after each call to the init()-function. This saves unneccessary updates, which cost time.

On the other hand, the window "time courses" defines four views, three different ones onto the x-variables (potentials), and one on the spikes z. The potentials are displayed as an IMAGE of size 10×10 (just for demonstration), a RASTER which displays the potentials over time as a grey-level plot, and a GRAPH, which selects a single potential trace and plots it as a function over time. The spikes are, finally, also plottet as a RASTER, ie., the values of the whole array are displayed over time. More about this later, when we look at the actual graphical output (Figure 2.3 for the impatient).

2.3.2 Coupling of Parameters and Panel Controls

The views on variables defined in the previous section allow to observe in real time variables of the simulations. However, we might also want to change parameters and see where that leads to. To do this we have to add control elements to the main window of the GUI (e.g., 2.1). These elements can then be coupled to parameters of the simulations.

There are two types of control elements available in Felix: Switches and Sliders. Switches are represented by buttons; they can be ON or OFF, and thereby they can "switch" code execution between alternative segments (Switches are not used in this section, but see section 3). The second control element are Sliders. These can take values in a whole range and can thereby represent continuous parameters of the simulations.

How does this work inpractice? Let us assume we want to control the parameters I, J_0 and σ in the simulation of the leaky-integrate-and-fire network. These are the global input, the global effective coupling strength, and the noise level. For each of these we have to define a new variable of type SliderValue (the reason for this follows soon). These new variables we have to embed in the GUI, and we can use them in the simulation code as well. The code below shows how this is achieved.

```
SliderValue sI     = 100;
```

```
SliderValue sJ0    = 50;
SliderValue ssigma = 0;

BEGIN_DISPLAY

  SLIDER( "input",     sI, 0, 200)
  SLIDER( "coupling", sJ0, 0, 200)
  SLIDER( "noise", ssigma, 0, 100)

  WINDOW("time courses")

  ....

END_DISPLAY
```

sI, sJ0 and ssigma are the new variables of type SliderValue. The SLIDER()-macros then add the slider to the GUI, giving them names and certain lower and upper bounds. This code-snippet reflects one problem with Sliders: Constrained by the XWindows/XView system they can only take integer values. In the example these ranges are $\{0, 1, 2, \ldots, 200\}$ for input sI and coupling $sJ0$, and $\{0, 1, 2, \ldots, 100\}$ for the noise level $ssigma$. Accordingly, when the variables are used in the code implementing the dynamic equations of the simulated system they have to be scaled appropriately. For instance, in the step()-routine we could have code like

```
for (i=0;i<N;i++)  // leaky integration for all neurons
  leaky_integrate ( tau, pot[i],
                0.01*( sI + sJ0*v1[i] + ssigma*gauss_noise() ) );
```

The factor 0.01 scales the ranges for sI , sJ0 into the intervals [0,2[and that for ssigma into the range [0,1[. This is somewhat uncomfortable, but one gets used to it quickly.

Finally, note that now that we have replaced the original variables I, $J0$, and σ by slider variables, we can delete their original declarations in the program. They don't appear in the code anymore, but instead they are controled by the graphical user interface, see Figure 2.2.

2.3.3 Running simulations using the graphical interface

Figure 2.2 displays the GUI after the control elements have been added. The display windows we have defined are still hidden. We can open them by right-clicking the "Windows"-button, which pops up a list of all availabe windows.

Figure 2.3 shows the interface after the display windows have been opened and placed on the screen. On top of the control panel is shown the coupling matrix and to the left of both the window "time courses" containing the simulation variables.

By left-clicking the "Environment"-button this configuration can be saved such that the GUI comes up in the same state the next time the program is started again (right clicking the "Environment"-button gives some more options). This automatic loading of parameters from a default environment file overrides any explicit initialisations of slider variables possibly done in the source code.

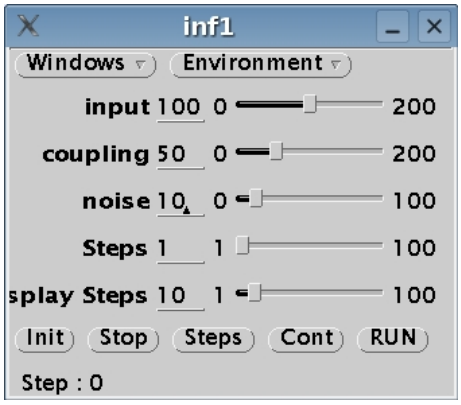

Figure 2.2: Graphical user interface after adding sliders for parameters of the simulation.

The Felix example directory contains the code of a leaky-integrate-and-fire network with GUI and file output. You might want to experiment with it, before proceeding to the next section which describes how file output is declared. In particular, note that the label on top of each view is clickable and brings up control panels for the grey-scales of images and rasters, or the selected variable index in graphs that display arrays.

2.4 Adding Output of Data

Analogous to MakeDisplay() which defines the graphical output, the function MakeOutput() defines output that is supposed to go to files. The macro NO_OUTPUT can be used if no such output is required. The macros BEGIN_OUTPUT and END_OUTPUT in turn enclose code for data output. This defines a function MakeOutput() which is called just after the initialisation, and after each subsequent simulation step. It is possible to select temporal sub-ranges for output only as well as subsets of arrays; this is explained in detail in chapter 3. The following code shows how variables of the leaky-integrate-and-fire model can be saved.

```
BEGIN_OUTPUT

  OUTFILE("potentials")
    SAVE_VARIABLE( "pot", x, VECTOR, N, 0, 0, 0, 0 )

  OUTFILE("spikes")
    SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON )
    SAVE_VARIABLE( "out", z,  bVECTOR, N, 0, 0, 0, 0 )

END_OUTPUT
```

Two output files are defined, “potentials” and “spikes”, with obvious meaning. An arbitrary number (up to operating system constraints) can be opened, each of which can save a number of variables per step. [Those are stored in sequential order, which might cause problems when the data has to be reread in data-analysis programs, because of the possibly complicated record structure. It is probably more convenient to store only one variable per file as in the shown example.] The

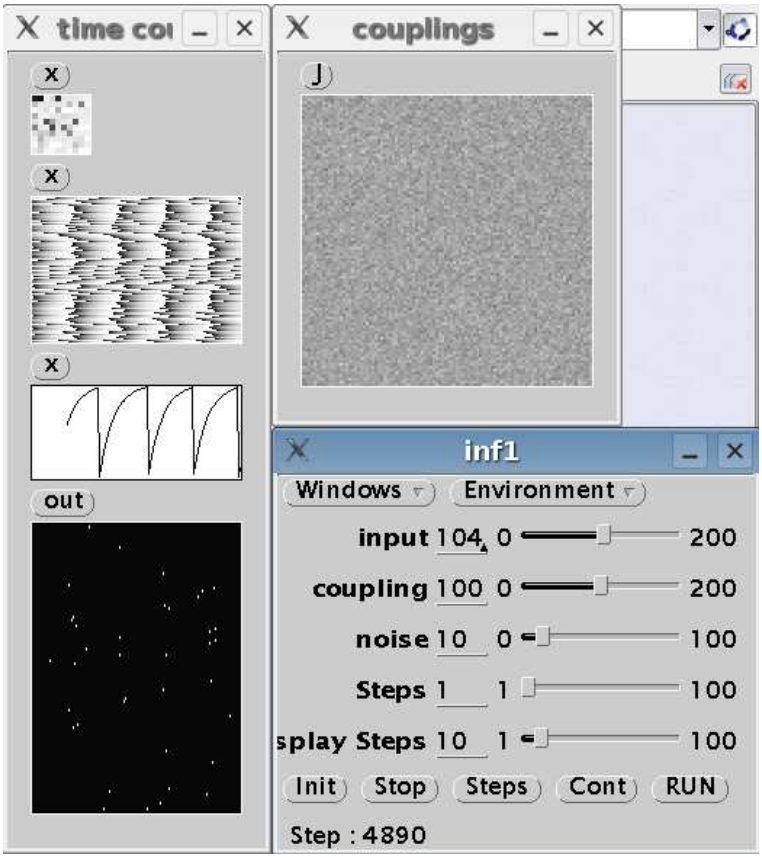

Figure 2.3: User interface showing the control panel and the two windows created for displaying dynamic variables.

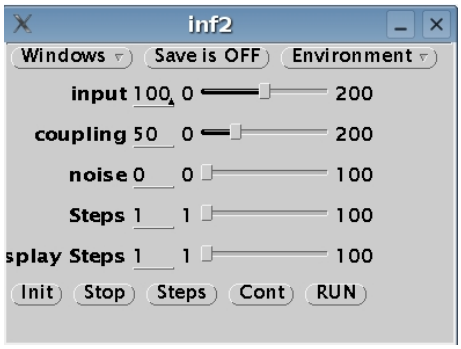

Figure 2.4: Control panel of the graphical user interface after output files have been declared in the program.

variable x is stored as a vector of length N to the file "potentials", whereas z is stored as a binary vector of length N to the file "spikes".

By default, data is saved in binary format, so that it would not be readable by humans, but save space. The default is changed for the second file in the example, where a flag is set for (human-readable) ASCII output.

Figure 2.4 shows the main GUI window after file output has been defined. The new button "Save is OFF" indicates that the ouput has not yet been activated. If it is pressed, file output starts. If it is

pressed repeatedly during a simulation, the state of the button toggles, and data generated during the activated phases are appended to the output files. Right-clicking on the "Save is OFF"-button brings up further options, not all of which are fully implemented, nor well tested.

Have a look into chapter 5 (or the example programs, or the source code of output.c/h) for possibilities to select sub-ranges of array variables for output. This can be useful for large simulations, because otherwise the amount of generated data can quickly become tremendeous.

Chapter 3

Graphical User Interface

Chapter 2 presented a brief example of how Felix programs are structured and what the main properties of the Graphical User Interface are. The present chapter looks into the GUI in more detail.

In general each Felix program has a main window with control elements for running a simulation and manipulating its parameters in real-time. Switches are binary (ON/OFF) controls that allow for a conditioned execution of code segments. Sliders in contrast can take values in a range of numbers and can therefore be used to set parameters of a simulation. Beside the main window a Felix simulation in general will have one or more display windows. These can contain graphics objects of various types, which display views on variables as, e.g., graphs, functions, images, or plots. Finally each Felix simulation has an "Environment" allowing the user to store and load parameter settings in external files. The present chapter will go through the mentioned components step by step.

3.1 Creating a GUI

Each Felix application has to define which objects (variables, vectors, matrices ...) are displayed on the screen and how this shall be done. For this a function

```
void MakeDisplay(){...}
```

has to be supplied which contains definitions of the graphics objects to be displayed. The function-body of "MakeDisplay" can be empty, if no graphical output is needed. In that case a basic main window is still generated, see Figure 2.1, but no display windows. There are three Macros that support the definition of the interface

```
#define BEGIN_DISPLAY  void MakeDisplay(){
#define END_DISPLAY    }
#define NO_DISPLAY     void MakeDisplay(){}
```

Beside a number of buttons to initialialise, run, stop, and resume a simulation, the each GUI by default contains two sliders "Steps" and "Display Steps". These control the display and multi-step

mode of a simulation. "Display Steps" sets the interval in simuation steps at which the graphics objects in the display windows are updated. "Steps" in contrast sets the number of steps that are executed in multi-step mode (i.e., after stopping an initialised simulation) when the Steps-button of the GUI is pressed. The maximum steps of both these sliders by default is 100, which is convenient for most situation. Should it be necessary, the numbers can be changed using the macros MAXSTEPS() and MAXDISPLAYSTEPS() in the definition of MakeDisplay().

```
MAXSTEPS( steps )
MAXDISPLAYSTEPS( steps )
```

In very new versions of Felix the colormap for the variable views can be selected by using the macro `COLOR_MAP( map )` somewhere at the top of the display declaration. Possible values for "map" are:

CMAP_BW : The default gray-scale map; black: low-values; white: high values

CMAP_RED : Black to crimson-red intensity coded (quite hellish)

CMAP_GREEN : All green (looks like the aliens are around the block)

CMAP_BLUE : All blue (deep not light blue)

CMAP_RAINBOW : Blue - green - red colour scale (quite fancy)

In the non-gray maps the lowest and highest values are black and white, respectively. This makes clipping at range boundaries nicely visible.

3.2 Simulation Control Elements

3.2.1 Switches

Switches are logical flags, that may be used in a simulation to interactively select execution of different parts of the source-code.

A switch must be globally declared as a variable of type 'SwitchValue' on the top of the application-source file.

A switch can be ON or OFF:

```
#define OFF 0
#define ON  1
```

To create a button in the main-window that affects the switch-variable the function MakeSwitch() or the Macro SWITCH() must be called in MakeDisplay() :

```
#define SWITCH(name, var) MakeSwitch(name, &var);
```

Here "name" is a string that appears on the switch-button in the GUI and "var" is its accociated variable of type SwitchValue.

If a user wants to change values of switch-variables at the source-code level the function SetSwitch() or the Macro SET_SWITCH() MUST be used. Simply assigning a value to a switch-variable is not enough, because the new value will not be signalled to the XWindows-system, such that the state of the switch would be no longer represented by its corresponding button.

```
#define SET_SWITCH(var, value) SetSwitch(&var, value);
```

"var" is the variable to be set; possible values are ON or OFF.

Example:

```
...

SwitchValue sw = OFF;          /* define the switch-variable */
...

BEGIN_DISPLAY
...
SWITCH( "this-or-that", sw)  /* define the switch-button    */
...
END_DISPLAY



int void step()

  ....

  if (sw)                      /* execute code depending on
                                  the state of sw.            */
  {     /* do this */
    ...
  }
  else
  {     /* do that */
    ....
  }

}
```

3.2.2 Sliders

As switches sliders are used to interactively control a running simulation. The difference is, that they are multi-valued and thus may be chosen to influence parameters of the model. To create

a slider the user has to declare a global variable of type 'SliderValue' (i.e., int). This variable is associated with a graphical slider in the GUI by a call to the function MakeSlider() or the macro Slider() inside the initialization routine MakeDisplay().

Sliders appear in the main-window in the order of their declaration in MakeWindow(). The macro SLIDER_COLUMNS(columns) can be used to arrange them in more than 1 column (default).

```
#define SLIDER(name, var, min, max)  MakeSlider(name, &var, min, max);
```

"name" is a string that appears to the left of the slider. "var" is the variable of type SliderValue that stores the current value of the slider. "min" and "max" set the range allowed for changes in the sliders value.

To set or change slider-values at source-code level the function SetSlider() (or macro SET_SLIDER) must be used:

```
#define SET_SLIDER(var, value) SetSlider(&var, value);
```

"var" is the name of the slider-variable and "value" the new slider-value of type SliderValue (i.e., int).

Unfortunately, XView restricts sliders to integer-values. Thus, if an application contains floating-point parameters which shall be modifiable from the graphical interface, one has to scale the corresponding slider-values to the appropriate range.

Example :

```
SliderValue param = 50;    /* define and initialize a slider-variable */
...


BEGIN_DISPLAY
...
SLIDER( "parameter", param, 0, 100)  /* generate an instance of a
                                        XView-slider in the main window
                                        associated with variable "param"
                                        The name of the slider is
                                        "parameter", its range [0,100]
                                     */
...
END_DISPLAY



int void step()
{
  float float_param;
```

```
  ..

  float_param = .01*param;       /* this casts the slider-value to float
                                  * in the range [0,1.0]. Observe, that
                                  * only 100 different values are possible!
                                  */

  ..

  if (any_condition)
    SET_SLIDER( param, 50 )      /* this sets the slider to a well-defined
                                  * value (here 50).
                                  */

}
```

3.2.3 Timer

Usage of the Macro TIMER() in the definition of MakeWindow() creates an extra slider which influences the time between two successive simulation-steps.

```
TIMER( max )
```

The timer slider will have a range from 0 to max. If the value is zero the timer is off. Otherwise it effects the times between calls to the step() routine in a running simulation. The value in principle is supposed to be in Milliseconds, but this shouldn't be taken too seriously.

3.3 Display Windows and Views on Variables

3.3.1 Display Windows

The Macro WINDOW() or function MakeWindow() create a new window for graphical output. The string "name" appears at the top of the window and in the Window list of the main control window.

```
#define WINDOW(name) MakeWindow(name);
```

The WINDOW-statement must be called in MakeDisplay() before any other output can be directed to the screen, i.e. before any graph, image, raster, or other variable views are defined.

Several windows can be defined by repeated calls to WINDOW(). In this case the last declared window is always the active one, meaning that subsequently declared graphics objects are placed into that window.

All window-names are collected into the "Windows"-menue at the top left of the main control window. If a window is closed selecting it from this menue will reopen it.

3.3.2 Views

After a Window has been defined it can be filled with graphical views on simulation variables (images, graphs, etc). The functions to create the various possible views all have a similar structure. Consider, e.g., creation of an image by using the macro IMAGE():

```
IMAGE(name, row, col, var, type, dim_x, dim_y, min, max, zoom)
```

The first argument is the "name" of the view. It will appear on a button on top of the view.

"row" and "col" are two arguments to control positioning of the view in the display window. This is covered in the next subsection 3.3.3.

"var, type, dim_x, dim_y" then characterise what is actually displayed. The type of a displayed variable "var" must be declared and, if it is a vector or an array, also its dimensions. Possible display types are described below in subsection 3.3.4.

The last argument of a view definition, "zoom", is a (small) integer number that controls how big a view appears on screen. Default value is 1. In that case, e.g., each entry of a matrix-valued variable will be displayed by one pixel in a rectangular image. Larger numbers for zoom correspond with more pixels and bigger images.

3.3.3 Placement of Views inside a Window

There is a simple mechanism to control positioning of view elements.

The 2cd and 3rd arguments of a view-definition are coordinates for the upper left corner of the view. These can be given directly by specifying raw pixel coordinates.

Alternatively, each display window can be considered as being partitioned into a coarser rectangular grid. Several macros support placing views in that coarse grid.

R0, AR and NR can be used as values for the 2cd, and C0, AC, and NC as values for the third argument of a view defining function.

R0 and C0 specify the first row and column, respectively.

AR and AC specify that the view has to be placed in the actual row or column, respectively.

NR and NC specify that the view has to be placed in the next row or column, respectively.

Example (cf., section 2.3)

```
BEGIN_DISPLAY

  WINDOW("time courses")

    IMAGE(  "x", AR, AC, x,  VECTOR, 10, 10,  0.0,  1.0, 4)
    RASTER( "x", NR, AC, x,  VECTOR,  N,  0,  0.0,  1.0, 1)
    GRAPH(  "x", NR, AC, x,  VECTOR,  N,  0, 0, 0, -.01, 1.01 )
```

```
    RASTER( "z", NR, AC, z, bVECTOR,  N,  0, -.01, 1.01, 2)

  WINDOW("couplings")

    IMAGE(  "J", AR, AC, J, CONSTANT MATRIX, N, N, -4./N, 4./N, 4)

END_DISPLAY
```

This example defines two windows with names "time courses" and "couplings". The first window contains four graphics views, the second only 1. In both cases the first view is placed at position (AR, AC), the actual row and actual column, which by default after creating a new window with WINDOW() is equal to (R0, C0), the upper left location in the coarse grid. The subsequent views in the first window are then placed at (NR, AC), meaning the next row but actual column. Therefore, the four views are placed in a single vertical column. In contrast, replacing (NR, AC) by (AR, NC) in the code would place the views all in a horizontal row, and (NR, NC) places them along the diagonal of the coarse grid (which wouldn't look too nice).

3.3.4 Types of Display Variables

Felix supports displaying of variables of three base types: char, int, and float. A fourth type, packed bits, is obsolete and shouldn't be used. Displaying double, long, and unsigned variables is not supported.

Variables can be scalars or vectors / arrays. There are several type macros that can be used in the view display type definitions:

```
#define CHAR_TYPE      0x02
#define INT_TYPE       0x04
#define FLOAT_TYPE     0x08

#define ARRAY_TYPE     0x20

#define ARRAY_CHAR_TYPE    (ARRAY_TYPE | CHAR_TYPE)
#define ARRAY_INT_TYPE     (ARRAY_TYPE | INT_TYPE)
#define ARRAY_FLOAT_TYPE   (ARRAY_TYPE | FLOAT_TYPE)
```

The basic display types above are conveniently redefined in some of the Felix libraries, e.g.:

```
vector.c/h : VECTOR,  MATRIX  = ARRAY_FLOAT_TYPE
             bVECTOR, bMATRIX  = ARRAY_CHAR_TYPE
nn.c/h :      LAYER            = ARRAY_FLOAT_TYPE
            SPIKE_LAYER        = ARRAY_CHAR_TYPE
```

It is sometimes desired not to provide just a variable to a view, but a pointer to a variable. The variable the pointer references can then change dynamically in every display step. The POINTER-macro sets the respective type bit.

```
#define POINTER         0x8000
#define TO              |
#define CONST_BIT       0x4000
#define CONSTANT        CONST_BIT |
```

A variable can be declared CONSTANT if it does not need to be updated during a running simulation. A CONSTANT variable is updated only after a call of the init()-function at the beginning of a simulation, ie., by pressing the Init- or Run-buttons of the GUI.

Examples

```
BEGIN_DISPLAY

  WINDOW("time courses")

    IMAGE(  "v", AR, AC, v, MATRIX, 10, 10,  0.0,  1.0, 4)
    IMAGE(  "z", AR, NC, z, CONSTANT MATRIX, 10, 10,  0.0,  1.0, 4)
    IMAGE(  "y", AR, NC, y, POINTER TO bVECTOR, 10, 10,  0.0,  1.0, 4)
    IMAGE(  "x", AR, NC, x, POINTER TO CONSTANT VECTOR, 10, 10,  0.0,  1.0, 4)
    ...
```

The first IMAGE defines a view on a MATRIX (ARRAY_FLOAT_TYPE) of size 10×10. This is how a view definition usually declares a variable type. The second IMAGE also defines a view on a MATRIX , but because the matrix is declared CONSTANT the graphics view will only be updated when the simulation starts. The third IMAGE refers to a binary vector image (bVECTOR = ARRAY_CHAR_TYPE). However, a pointer to the variable y is declared so that the array y may change dynamically. The fourth image also declares the variable z a pointer, but this time a constant one, so that it could change where it points at, but its view is refreshed only at the beginning of a simulation. (I can't remember I ever used this possibility during the last 15 years).

3.3.5 Image-Views

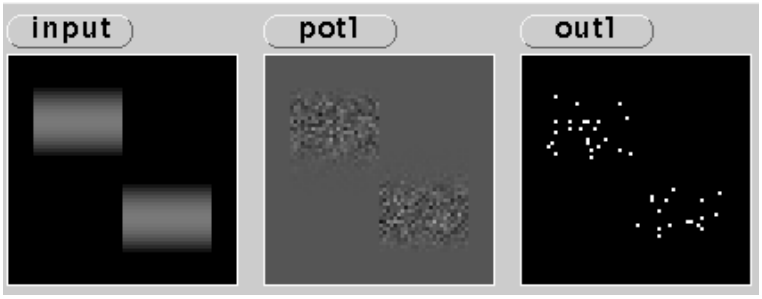

Figure 3.1: 2d grey-scale images of 3 arrays in a neural field model. Left: input; middle: potentials; right: spikes of the cells in the model.

An image is a two-dimensional grey-scale plot of the current state of a two-dimensional variable. It is created by calling the macro IMAGE() inside the function MakeDisplay():

```
IMAGE(name, row, col, var, type, dim_x, dim_y, min, max, zoom)
```

Here "name" is a string that appears on a button above the image. "row" and "column" give the position of the image in the currently active window, see section 3.3.3.

"var" is the variable to display as an image, "dim_x" and "dim_y" its dimensions, and "type" its display type as described in section 3.3.4.

The floating point variables "min" and "max" set the grey-scale of the image and should be set to the expected min- and max-values for the variable.

The integer valued argument "zoom" sets the size of the image. Each element of the variable is displayed as a square of size zoom×zoom.

Vectors, i.e. one dimensional arrays, can be displayed as images, too, by providing appropriate dimensions in the call to IMAGE(). If dim_x*dim_y is bigger than the actual vector size, the behaviour is undefined and core dumps can potentially occur. If it is smaller the remaining components are not displayed.

3.3.6 Raster Plots

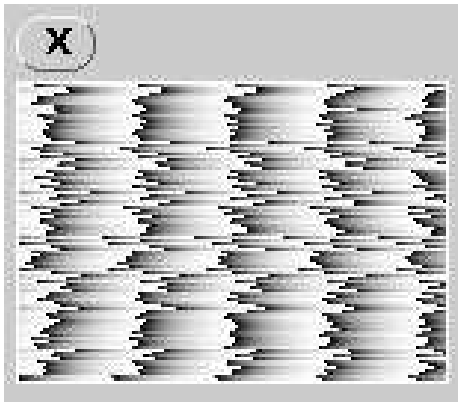

Figure 3.2: Raster plot of the grey-scale-coded potentials of 100 leaky-integrate-and-fire neurons over time.

A raster plot displays a vector or 2d-array as a function of time. Each component of the variable is shown grey-level coded on a seperate line.

```
RASTER(name, row, col, var, type, dim_x, dim_y, min, max, zoom)
```

The arguments are the same as in image. If "dim_y" is not zero it is assumed that "var" is a 2-dimensional array that has to be interpreted as a vector of length dim_x*dim_y.

"zoom" only sets the height of each displayed line (in pixels). It has no influence on the number of time-steps that fit on one line.

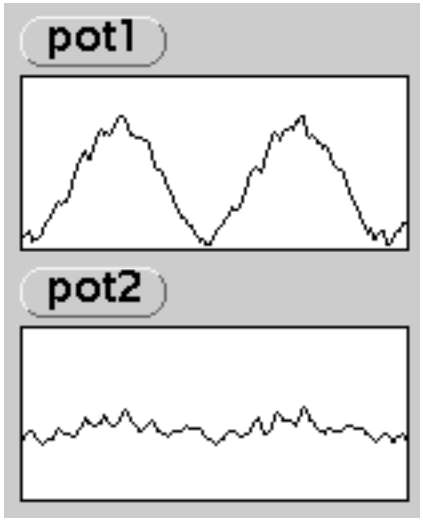

Figure 3.3: Two single variable graphs over time

3.3.7 Single Variable Graphs

A graph displays a single scalar variable, or a component of a 1 or 2d-array as a function of time.

```
GRAPH(name, row, col, var, type, dim_x, dim_y, x, y, min, max)
```

All but the "x" and "y" parameters are the same as in IMAGE.

In case of ARRAY_TYPES (see section 3.3.4) "x" and/or "y" specify which component of the variable has to be displayed initially.

3.3.8 xy-Plots

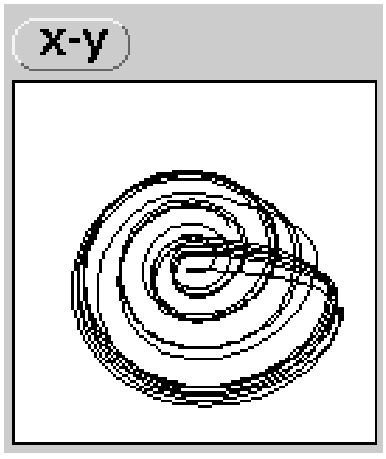

Figure 3.4: Example showing an x-y-plot of two variables of a Roessler oscillator

This view object displays an xy-plot of two variables. The variables may be independently chosen as single scalars or components of 1 or 2d-arrays.

```
PLOT(name, row, col, var1, type1, dim_x1, dim_y1, x1, y1, min1, max1,
                     var2, type2, dim_x2, dim_y2, x2, y2, min2, max2,
                     zoom)
```

All parameters are the same as in GRAPH, but note that one has to specifiy two variables with adjoined information about variable type, the subelement to select from arrays, and the grey-scale settings.

3.3.9 Arrays of Images

This type of view mainly aims at displaying 2-dimensional arrays of 2-dimensional images as they arise, e.g., in neural field models, where each local unit in a 2d-field has an individual 2d-lateral connection kernel. 1-dimensional arrays of 2-dimensional images (e.g., a stack of cortical layers) can also be displayed.

```
IMAGE_ARRAY(name,row,col, var,t, dim_x, dim_y, d_x, d_y, x, y, min,max,zoom)
```

"name, row, col, min, max, zoom" have the same meaning as usual (see IMAGE).

"var" and "t" define the variable and its display type (which must be an ARRAY_TYPE (usually MATRIX) and can be a POINTER type)

"dim_x" and "dim_y" define the dimensions of the array of images. If "dim_y" is zero, but "dim_x" positive a one dimensional array of images is assumed.

"x" and "y" specify which of the sub-images of the array of images is displayed initially (can be overridden by the Environment).

"d_x" and "d_y" define the size of the displayed images in x and y direction. They must both be positive.

3.3.10 Functions

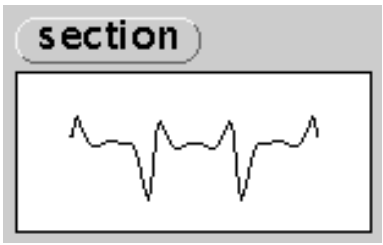

Figure 3.5: A function view

A function view plots a one-dimensional array (VECTOR) as a function of the array index. Single functions , and one- and two-dimensional arrays of functions are possible (cf., IMAGE_ARRAYS).

```
FUNCTION(name, row, col, var, type, points, dim_x, dim_y, x, y, min, max)
```

"name, row, col, var, type, min, max" are the same as in GRAPH.

"points" is the number of data points in any individual array that is to be plotted as a function.

"dim_x, dim_y, x, y" are used for arrays of functions. If "dim_x" is bigger than 0 and "dim_y" is zero a one-dimensional array of functions is assumed; if they are both bigger than 0, a two-dimensional one. "x" and "y" define the initially selected function to display (can be overridden by the Environment).

3.4 View Settings Frames

Each graphical view object in a window has a "settings frame" associated with it that can be used to control the grey-scale ranges of the view and to select sub-elements in case of array variables for graphs, functions, or image arrays. The settings frame pops up, if the button of a view showing the view's name is pressed.

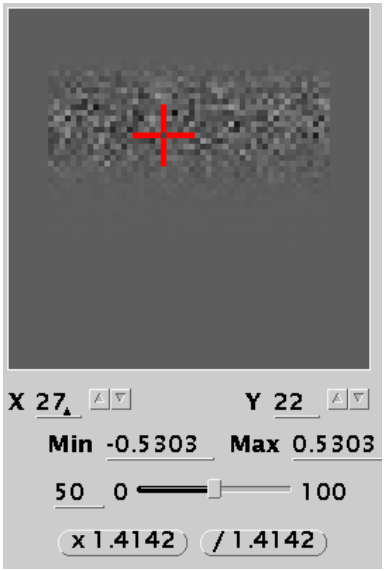

Figure 3.6: A two-dimensional settings frame

Figure 3.6 shows a two-dimensional settings frame as it could occur for a graph of single element of a 2d array. The frame shows the full array as a grey-scale image where the particular element to display is indicated by the red crosshair. The element can further be selected by the x and y textfields. If a variable has to be selected from a one-dimensional array only, the 2D-image is replaced by a slider. If the variable to display is a scalar, no extra element selectors will appear in the corresponding settings frames, but only the controls for setting the grey-level.

The definitions of all views contain arguments "min" and "max". These set the initial grey-scale for that view. They can be changed in the settings frame, too. If the scale is changed, the new settings can be stored to an environment file (see section 3.5).

3.5 Loading and Saving GUI Settings

The Felix GUI for convenience provides the possibility to load and store settings of the graphical interface. The "Environment"-button on the main control window serves this task.

Right-clicking the "Environment"-button brings up a menue with four options.

Save This saves the current settings in a default file

Load This loads settings form the default file

Save as .. This pops up a window where the current settings can be stored in an arbitrary file

Load ..] This pops up a window where settings can be loaded from an arbitrary file

Left-clicking the "Environment"-button by default saves the current settings in the default file.

The default file is located in a sub-directory "env" of the current working directory, ie., the directory the executable is located and run in. The default file has the same name as the executable.

The default file does not exist until it is created (by left-clicking the "Environment"-button). If the environment directory "env" does not already exist, it is created, too.

If a default file exists it is automatically loaded when an application starts. This overrides any explicit initialisations of switch or slider values, image grey-scales, or sub-selections in views that can plot array objects. Occasionally this behaviour is unwanted,. You then need to rename or delete the default environment file in the "env"-subdirectory.

Note: Changing the number of graphic objects (switches, sliders, windows, views) in the GUI definition of an application typically invalidates the environment file(s). Instead of using the file, the application will print an error message on the screen. This is sometimes uncomfortable for complex applications, because all settings have to be set anew. It can then be easier to hand-edit the environment files: If proper entries for the new (or deleted) objects are added, the file can be used again.

Chapter 4

Libraries

Felix comes with a number of function packages / libraries suitable for tasks often encountered in the modeling of neural networks and dynamical systems. The present section provides an overview.

4.1 Outline: Pools and Fields

Although not restricted to them, two types of models have been in the main focus during the design of Felix - networks comprising homogeneuos neural pools and layered, topographically ordered neural fields, cf., Figure 1.2 in section 1.2.

Given a single neural pool of N neurons its dynamics could be described mathematically by

$$\tau\phi_i(t) = -\phi_i(t) + I_i(t) + \sum_{i=1}^{N} w_{ij} f(\phi_j(t)) + \sigma\eta_i(t) \tag{4.1}$$

Here, the cells are modelled by a single variable for their membrane potentials, $\phi_i,\ i = 1, \ldots, N$, and by a graded sigmoid output or rate-function f. Single units are identical: they obey the same mebrane low-pass dynamics with time-constant τ and have the same rate-function. They might, however, receive different inputs $I_i(t)$ and noise $\eta_i(t)$ of strength σ, and their synaptic weights $w_{ij},\ j = 1, \ldots, N$ will differ. More complicated single neuron models are of course possible. Cells, in general, also don't need to be identical.

The dynamics of a single neural field in contrast can be written as

$$\tau\phi(x,t) = -\phi(x,t) + I(x,t) + \int w(x,x') f(\phi(x,x',t)) + \sigma\eta(x,t) \tag{4.2}$$

In contrast to (4.1) cells do not just have indexes, but a continuous spatial location x (which will, of course, typically be discretised in computer models). Units at one location interact only with neighbours nearby. This is reflected by the synaptic kernels $w(x,x')$ in (4.2). Beside this, the meaning of the symbols in equations (4.1) and (4.2) are the same.

Both kinds of models need similar construct to define and simulate the single units they consist of, e.g., the dynamics of the membranes ϕ and the output type of the units. Both, (4.1) and (4.2) above use first order low-pass filters and graded output by means of nonlinearities f. The main difference concerns their connectivity patterns. In pool-models all cells in one pool can potentially

reach all cells in the same or another pool – matrix-vector operations are most convenient to implement this kind of model, see section 4.3 below. Neural fields on the other hand reveal topographic neighbourhood structures - Felix provides constructs for the implementation of this kind of "integro-differential equation", too, see section 4.5.

Delays further play an important role in many neural models. They are supported in Felix by a container class that stores model trajectories over time and a number of fundamental routines to access delyed variables in simulations. There are in particular delayed convolution functions, that are needed if lateral propagation speeds in a field model are finite. Details can be found in section 4.6.

Noise is omnipresent in neural systems and in many other physical systems, too. In (4.1) and (4.2) noise inputs into the system is, for instance, represented by the processes η. These are commonly assumed independent and identially distributed Gaussian white noise with mean 0; σ sets the standard deviation. Other choices are Poisson processes of some rate which would reflect the spiking nature of inputs to neurons. Felix has a built-in pseudo-random number generator described in section 4.7.

Felix also has libraries with some numerical and image processing routines. Because, these are not well developed, they will not be described in this document.

4.2 Some Low-level Definitions

If not defined already in system headers, the Felix headers define the following macros

```
# define TRUE 1
# define FALSE 0

# define MIN(a, b) ((a) > (b)? (b) : (a))
# define MAX(a, b) ((a) > (b)? (a) : (b))
```

4.3 Matrix and Vector Operations

The Matrix/Vector functionality is a central part of Felix. Two base-types for variables are in general supported. Most Felix functions operate on scalars, vectors, or matrices of those.

BaseType : floating point values (for historical reasons these are C-type "float"; I don't want to go into the mess if changing to "double").

bBaseType: binary (0/1) values. One bit stored per memory-byte (unsigned char).

BaseType is used for all kinds of continuous cell variables, whereas bBaseType is useful for the representation fo binary vectors of "spikes".

(The bitBaseType available in early versions of Felix is obsolete and shouldn't be used. It used a packed binary format; one bit stored per memory-bit.)

Vector and Matrix-types are derived from the base-types

```
typedef BaseType * Vector;
typedef bBaseType * bVector;

typedef BaseType * Matrix;
typedef bBaseType * bMatrix;
```

Note: Matrices are internally stored as linearized arrays of rows in memory (ie., not as vectors of pointers to rows or columns).

4.3.1 Operations on Scalar Variables

```
BaseType leaky_integrate( float tau, BaseType v, BaseType expr )
```

This macro implements a simple Euler-Scheme for simulating leaky-integrator membranes: $\tau\frac{dv}{dt} = -v + \text{expr}$. Integration stepsize is set with SET_STEPSIZE(dx) and should be chosen such that dx/tau is small compared to 1. The variable "step_size" can be used explicitly in code if required. In conjunction with the later explained fire_reset()-function, "leaky-integrate and fire neurons" are straightforward to implement (see the example program inf.c).

Several basic nonlinear functions are available as rate-functions or for other purposes. They all take a single float as argument and return a single floating point value.

triangle(x) : $f(x) = 1 - |x|$ if $|x| < 1$ and 0 if $|x| >= 1$

rectangle(x) : $f(x) = 1$ if $|x| <= .5$ and 0 if $|x| > .5$

gaussian(x) : $f(x) = exp(-4 * ln(2) * x * x)$ (The factor 4*ln(2) ensures f(.5) = .5)

fermi(x) : $f(x) = 1/(1 + exp(-4 * x))$ (The factor 4 ensures df/dx(0) = 1.)

ramp(x) : $f(x) = 1$ if $x > 1$, 0 if $x < 0$ and x else

lin(x) : $f(x) = x$

tlin(x) : $f(x) = x$ if $x > 0$ and 0 if $x <= 0$

tquad(x) : $f(x) = x * x$ if $x > 0$ and 0 if $x <= 0$

4.3.2 Memory Allocation Routines

Before use, any vector or matrix-variable must be allocated. This is usually done in the top-level function main_init(). It *must* be done there if the variable is accessed for display in the graphical user interface. Use the following template for functions to allocate vectors and matrices:

```
<var_type> var;
var = Get_<var_type>( <dims> );
```

<var_type> stands for "Vector", "bVector", "Matrix", or "bMatrix". If a Vector-Type is supplied <dims> is the requested length. In case of a matrix <dims> = "rows, columns".

Allocated memory-space should be set free if no longer need. This is done by macros of the type

```
Free_<var_type>(var);
```

or simply by a call to the system-library function free($< var >$).

Example:

```
Matrix m = Get_Matrix(10, 10);      /* allocate memory for m */
 ......
Free_Matrix( m );                /* or alternatively: free(m); */
```

4.3.3 Cleaning Vectors and Matrices

All entries of a Vector or Matrix are set to zero by one of the following macros:

```
Clear_<var_type>(<dims>, <var>) .
```

For example:

```
Clear_bMatrix( rows, columns, m );
Clear_Vector( length , v );
```

4.3.4 Access to Elements of a Matrix

```
elem( m, i, j, columns )
```

This is a macro that gets or sets the element $m[i][j]$ of the Matrix or bMatrix m. 'columns' is the number of columns of m. (If i is set to 0 the macro can be used for VectorTypes, too.)

For example:

```
elem( m, 5, 6, 10) = 3.14;  /* Set m[5][6] to 3.14 */
x = elem( m, 5, 6, 10);     /* set x to m[5][6]    */
```

If you don't use this macro for matrices you have to keep in mind that matrices are stored serially in memory, i.e, elem(m, 5, 6, 10) would be equivalent to $m[5 * 10 + 6]$, but note that $m[5][6]$ *does not* work!

4.3.5 Raw I/O of Vectors and Matrices to/from files

Raw output in binary format to or from a stream is done with one of the macros:

```
Write_<var_type>( <dims>, <var>, stream );
Read_<var_type>( <dims>, <var>, stream );
```

These macros directly call the system-library functions fread() and fwrite(), thus they return the number of bytes actually read or written. For error-indications see fread() and fwrite().

For ASCII-output use the functions

```
Save_<var_type>( <dims>, <var>, stream );
Load_<var_type>( <dims>, <var>, stream );
```

Vectors and Matrices are stored row by row; b-Types are stored as sequences of zeros and ones. Entries are separated by blanks. On error the functions return -1; otherwise zero;

There are functions mainly for debugging purposes that print out Vectors and Matrices to stdout in the same manner as the Save-family does to files. These are

```
Show_<var_type>( <dims>, <var> );
```

4.3.6 Vector and Matrix Operations

Scalar Multiplication of two vectors of length n.

```
BaseType Skalar(int n, Vector v1, Vector v2)
BaseType bSkalar(int n, Vector v1, bVector v2)
int bbSkalar(int n, bVector v1, bVector v2)
```

Observe that the purely binary operation returns int-type.

Matrix-Vector Multiplication.

```
Vector Mult(int z, int s, Matrix matrix, Vector vector, Vector dest)
Vector bMult(int z, int s, Matrix matrix, bVector vector, Vector dest)
```

`dest = matrix * vector`, where "matrix" has z rows and s columns, "vector" has length s, and "dest" has length z. The functions return "dest". Observe that the purely binary operations return int-type, thus a vector to integers has to be supplied as 'dest' .

Maximum, Minimum, and Sum over Elements.

```
BaseType Sum(int, Vector);
int      bSum(int, bVector);

BaseType Max_Elem(int, Vector);
BaseType Min_Elem(int, Vector);
```

Norms and Scaling The following compute Vector Norms. (Induced) Matrix norms are not implemented at the moment, but matrices can be supplied to the functions below as well.

```
BaseType Vector_Norm_1(int n, Vector v);
BaseType Vector_Norm_2(int n, Vector v);
BaseType Vector_Norm_sup(int n, Vector v);

void Norm_Vector_1(int, Vector v, Vector out);
void Norm_Vector_2(int, Vector v, Vector out);
void Norm_Vector_sup(int, Vector v, Vector out);
```

The "Vector_Norm_" functions compute the 1, 2, and ∞- (or max- or sup-)norm of a vector, respectively (ie, the sum of absolute values, square-root of squares, or the largest absolute element). The "Norm_Vector_" functions first compute the norms and then scale the vectors to a norm of 1. They return the result in "out" which can be the same as "v".

The subsequent functions scale vectors and matrices or apply more general functions to each element

```
Vector Scale_Vector(int n, Vector v, BaseType offs, BaseType scale, Vector out);
Vector Vector_Apply(int n, Vector v, BaseType (*func)(BaseType), Vector out);
Vector Vector_Apply_Arg(int n, Vector v,
             BaseType (*func)(BaseType, void *), void *args, Vector out) );

Matrix Scale_Matrix(int z, int s, Matrix m, BaseType offs, BaseType scale, Matrix out);
Matrix Matrix_Apply(int z, int s, Matrix m, BaseType (*func)(BaseType), Matrix out);
Matrix Matrix_Apply_Arg(int z, int z, Matrix m,
             BaseType (*func)(BaseType, void *), void *args, Matrix out);
```

Scale_Vector and Scale_Matrix apply an affine transformation to the elements of the vector "v" or matrix "m". That is, they multiply all values by "scale" and add an offset "offs".

Vector_Apply and Matrix_Apply apply a user defined function "func" to all elements in the array. The user-defined function "func" takes a single float as input and returns a floating point value; the previously defined non-linearities can, e.g., be used. The function is, e.g., useful to compute the outputs of graded response neurons given their potentials and rate-function.

Vector_Apply_Arg and Matrix_Apply_Arg apply a user defined function "func" with more than a single argument to all elements in the supplied array. "func" takes a void pointer to a vector (or struct) of arguments and must return a single floating point value.

Results of the above functions are return in "out" which can be the same as the input array.

Setting / Changing whole Vectors and Matrices

```
void Set_Func_Vector(int n, Vector v,
                     BaseType (*func)(BaseType),
                     int shift, BaseType height, BaseType scale)
```

This function changes the vector "v" according to $v[i] += height * func((i - shift)/scale)$, where "func" is a scalar function. Note that the function is additive. It can be used to genrate shifted versions of vectors with certain profiles as hey appear as input stimuli in some neural networks.

```
void Make_Func_Band_Matrix(int n, Matrix J,
             BaseType (*func)(BaseType),
             BaseType height, BaseType scale )
```

This generates band-matrices with (row-)profiles given by a function "func". The function is additive. "height" and "scale" set the amplitude and width of the profile (e.g., if func is a Gaussian, scale would be the standard deviation)

```
void Make_Func_Band_Matrix_Cyclic(int n, Matrix J,
               BaseType (*func)(BaseType),
               BaseType height, BaseType scale)
```

As the previous one this function generates band-matrices with (row-)profiles given by a function "func", but wraps cyclically. The function is additive.

```
void Dilute_Matrix(int z, int s, Matrix m, BaseType p)
```

This function randomly sets entries in the matrix "m" to zero with probability "p". A Vector can be diluted by setting one of the size arguments to 1 and the other to the true lebgth of the Vector.

4.3.7 "Neural" Operations for Vectors and Matrices

"Sigmoid" output functions.

```
Vector Fv(int n, Vector vector, BaseType (*func)(),
          BaseType factor, BaseType threshold, BaseType width,
          Vector out)
```

Apply the function func() to all "n" elements of "vector". func() can be one of the scalar functions defined earlier in this section, or a user-defined one. Result are stored in the vector "out" (not neccessarily different from "vector"). "factor" and "width" may be used to scale the variable-values to the nonlinear range of func(); "threshold" sets an offset-value:

```
  out[i] = factor*func(  (vector[i]-threshold)/width  );
```

The function returns "out". If out equals NULL (or 0) on entry, an output array is allocated internally and returned by the function. The user has to free() the respective space, if it is no longer needed.

"Poisson" Processes

```
bVector ProbFire( int n, Vector v, bVector out)
```

Computes a n-dimensional binary random-vector from v, such that $prob[o[i] = 1] = v[i]$ and $prob[0[i] = 0] = 1 - v[i]$. It is not checked whether v[i] falls into the range [0,1]. The function returns "out". If out equals NULL (or 0) on entry, an output array is allocated internally and returned by the function. The user has to free() the respective space, if it is no longer needed.

Threshold Neurons.

```
bVector Fire(int n, Vector vector, BaseType theta, bVector out)
```

For all n elements of "vector" compare $vector[i]$ with a threshold "theta": set $out[i]$ to 1 if it is larger and to 0 otherwise. The function returns "out". If out equals NULL (or 0) on entry, an output array is allocated internally and returned by the function. The user has to free() the respective space, if it is no longer needed.

Fire-and-Reset Neurons.

```
bVector Fire_Reset(int n, Vecto vector, BaseType theta,
         BaseType reset, bVector out)
```

Same as Fire() but if $out[i]$ is set to 1 then $v[i]$ is reset to the value "reset". This function may be used to implement the "integrate and fire neuron model" (cf, example program inf.c). If out equals NULL (or 0) on entry, an output array is allocated internally and returned by the function. The user has to free() the respective space, if it is no longer needed.

4.4 Stimuli

Felix provides a number of "classical" stimulus functions like steps, ramps, rectangular, triangle-, and dirac-functions in the temporal domain as well as plane waves, discs, bars and Gabor patches in the spatial domain. Many spatio-temporal stimuli can be combined from these options.

An example program "stimuli.c" should be contained in the code-directory of this user guide.

4.4.1 Temporal Stimulus Functions

These function generate tims functions of a number fo common forms.

```
float TSine( float t, float T, float t0 );
float TRect( float t, float T, float t0 );
float TTriangle( float t, float T, float t0 );
```

```
float TSkewRect( float t, float T, float t0, float duty );
float TSkewTriangle( float t, float T, float t0, float duty );
float TSkewSine( float t, float T, float t0, float duty );

float TPulse( float t, float t0 );       // 1 for 1 time-bin
float TDiracPulse( float t, float t0 );  // mass 1 for 1 bin
float TStep( float t, float t0 );
float TRamp( float t, float t0, float slope );

float TInterval( float t, float t0, float t1 );

float TGaussian( float t, float t0, float scale );
```

Argument t in these functions is the simulation time (usually SIM_TIME), T are period durations, $t0, t1$ are offsets, temporal shifts or times at with events happen. For instance, in `TTriangle`, $t0$ is an offset, whereas in `TInterval` $t1$ is the time the stimulus switches on, and $t2$ the time it switches off again.

Most of these functions are scaled to a range [0,1] (including `TSine` for consistency).

The skew functions use different ON/OFF times for the rectangular function and rise/fall-times for the triangle and sine, respectively. "duty" is a number between 0 and 1 that determines the ON/OFF fraction.

`TDiracPulse` scales the return value by the inverse internal simulation time step (step_size) in order to obtain Dirac-pulses normalised to a mass of 1. `TPulse` returns a pulse of amplitude 1.

`TRamp` is the semi-linear function which is zero until $t0$ and increases with rate *slope* afterwards.

`TGaussian` is a temporal gaussian with maximum at t0. It can be used to simulate a smoothly rising and then decaying stimulus. The duration of the stimulus can be infcluence by the factor multiplicative *scale*.

4.4.2 Spatial Stimulus Functions

The following functions return common two-dimensional stimuli like bars and gratings. They are additive in order to allow for combinations. Therefore the user has to clear arrays explicitly as necessary.

In the following routines m is a matrix for the 2D-stimulus, w and h are its width and height, respectively. $x0$ and $y0$ are centre locations of stimuli. Because the functions are additive, complex stimuli can be constructed by centering several sub-stimuli at different locations. Argument *amplitude* is the amplitude of a component. It is possible to insert the temporal stimuli of the previous section here in order to obtain spatio-temporal stimuli.

```
SWholeField( Matrix m, int w, int h, float amplitude);

SRect( Matrix m, int w, int h, float x0, float y0,
         float w0, float h0, float phi, float amplitude);
```

```
SProfile( Matrix m, int w, int h, float x0, float y0,
            float w0, float h0, float phi, float amplitude,
            float scale, float (*func)(float));

SDisc( Matrix m, int w, int h, float x0, float y0,
            float d, float amplitude);

SCircularFunction( Matrix m, int w, int h, float x0, float y0,
         float amplitude, float scale, float (*func)(float) );

SPlaneWave( Matrix m, int w, int h, float amplitude,
            float k0, float phik,  // wave vector, amplitude and angle
            float psi );           // phase

SGabor( Matrix m, int w, int h, float x0, float y0, float amplitude,
        float sig0, float sig1,    // principal and second sigma
        float k0, float phik,      // wave vector, amplitude and angle
        float phi, float psi );    // angle(k,sig0) and temporal phase

SCenterSurroundGrating( Matrix m, int w, int h, float x0, float y0,
     float r0, float a0, float k0, float phi0, float psi0,
     float r1, float a1, float k1, float phi1, float psi1 );
```

`SWholeField` adds a homogeneous offset to the whole field.

`SRect` adds a rectangle at orientation *phi*, centre location $x0, y0$, width $w0$ and height $h0$. (The quality of the rectangle is not good. Use the next function for stimuli with less discretisation artefacts.)

`SProfile` as `SRect` adds a rectangle to the stimulus array but with a certain profile along the y-axis (if phi=0) specified by the function *func*. *scale* scales the argument of that function, that is, it sets its length-scale. Note that the function takes a float-argument and returns float. If standard functions like sin, cos, etc from math.h are desired they have to be wrapped, because they take double-arguments (More concretely: define a function `float myfunc(float x){...}` that just calls the function wanted and provide the new function "myfunc" as an argument to `SProfile`).

`SDisc` adds a disc of diameter *d*. (The quality of the disc is bad, especially for small discs. The next function might be useful to get more appropriate results.)

`SCircularFunction` adds a circular stimulus with radial profile *func*. *scale* scales the argument of the profile function multiplicatively.

`SPlaneWave` adds a plane wave with wave number $k0$ and phase shift *psi*. The angle *phik* defines the direction of the wave.

`SGabor` adds a Gabor-patch. The meaning of the arguments are given in the definition above. (The angle *phi* is the angle between the wave-vector and the first principle axis of the envelope gaussian. If the gaussian is circular symmetric this angle is arbitrary.)

`SCenterSurroundGrating` is a stimulus consisting of a sine-grating with parameters $a0, k0, phi0, psi0$ in an inner circle of size $r0$, and a second grating with parameters $a1, k1, phi1, psi1$

in the annulus from $r0$ to $r1$. $a0, a1$ are the amplitudes, $k0, k1$ the wave numbers, $phi0, phi1$ the wave directions, and $psi0, psi1$ the phases of the gratings.

Note 1: Most of these functions are not very quick and could be optimised. If only static stimuli are needed, it is probably a good choice to compute them only once in main_init or init.

4.4.3 Dynamic Stimuli

If moving graitings or gabor patches at a fixed location but with moving sinusoidal modulation are needed, it usually suffices to recompute `SPlaneWave` or `SGabor` per simulation step with sinusoidally modulated phases psi. In `SCenterSurroundGrating`, both the inner and outer grating can be made moving this way. The example program "stimuli.c" contained in the code-directory of this user guide shows some examples.

There are also some macros and functions that allow to move stimulus centres dynamically in various ways. These functions assume that a centre has a location and velocity given by 4 floating point numbers cx, cy, vx, vy. The following macros and functions can be used to update these variables which in turn can be used as location variables $(x0, y0)$ in argument lists of the stimulus construction functions in the previous subsection:

```
advance_centre(x,y,vx,vy)

jitter_centre_location(x,y,s)
jitter_centre_velocity(vx,vy,s)

centre_is_in_circle( x, y, x0, y0, r)
centre_is_in_rect(x,y,l,r,b,t)

void jitter_centre_direction( float *vx, float *vy, float s);
void bounce_centre_velocity(float *vx, float *vy, float nx, float ny);
```

The first 5 functions are actually macros (defined in stimulus.h); the last two are true functions (defined in stimulus.c).

`advance_centre` updates the location of a centre given its velocity and the current step_size

`jitter_centre_location`, `jitter_centre_velocity`, and `jitter_centre_direction` add gaussian white noise of standard deviation s to the location or velocity variables, where `jitter_centre_direction` renormalises the velocity afterwards (the normalisation algorithm is sub-optimal; small numeric errors can accumulate).

`centre_is_in_circle` and `centre_is_in_rect` are macros that return 1 if a centre is inside a given circle at location $x0, y0$ of radius r or a rectangle bounded by l, r, b, t (left, right, bottom, top) respectively; otherwise they return 0.

`bounce_centre_velocity(float *vx, float *vy, float nx, float ny)` reflects the centre velocity given a surface normal of (nx, ny). It is not checked whether the normal is truly normalised to 1.

The example program "dynamic_stimuli.c" contained in the code-directory of this user guide shows several examples, i.e., either a disk or oriented bar that 1) bounces back and forth, 2) moves

through the input area along random pathways, 3) bounces through the input area reflected at the boundaries, 4) performs a random walk, or 5) performs a 2-dimensional Ornstein-Uhlenbeck process around the center of the input area.

4.5 Field Models, Spatial Convolutions

Field models are two-dimensional, topographically arranged neural networks, which are typically only connected within certain neighbourhoods, see Figure 1.2.

Although Felix supports one-dimensional and two-dimensional fields, only two-dimensional ones are described in this document.

4.5.1 Kernels or Filters

As stated, cells in neural fields are connected only locally. Felix assumes rectangular connectivity regions, which are called Kernels, or Filters, or receptive Fields. The precise name chosen depends on the context and on the scientific community ("kernels" appear in integro-differential equations in Mathematics, "filters" in image processing algorithms in computer science, and "receptive fields" in neural networks – all three concepts are "very closely related" (to speak cautiously).

```
/* two-dimensional Kernels/Filters */

typedef BaseType * Kernel;
typedef BaseType * UniKernel;

typedef bBaseType * bKernel;
typedef bBaseType * UnibKernel;
```

The difference between Kernels and UniKernels is that in some neural fields all units have the same "filters" (think, e.g., of a layer of cells detecting orientation at a fixed orientation), whereas in others each cell has its own "receptive field" (e.g., in a full orientation tuning map). In the first case one would story only a single copy of the kernel (UniKernel/UnibKernel), whereas in the second case a field of kernels is required (Kernel/bKernel)

4.5.2 Correlation and Convolution Functions

Again somewhat depending on scientific community, operations envolving kernels in field equations are written as "convolutions" $\int k(x - x')f(x')dx'$ or "correlations" $\int k(x + x')f(x')dx'$. The main difference is just mirroring the respective kernel (here shift-invariant UniKernels). Felix implements both options. In neural field applications one would (probably) prefer correlations because they measure the similarity (correlation) of the (input) f with the kernel local at location x.

There are a pretty large number of correlation and convolution functions in Felix, which differ in the types of arguments, and how they deal with the boundaries of a field.

They all take a input field "in" of size x×y and a kernel (Uni or Multi) of size kx×ky; they all return a field "out" of size x×y.

If the local operations are correlations the base name of the function is "Correlate", and it is "Convolute" for convolutions.

If the input field is of binary type (e.g., a field of 0/1 spikes) a "b" is added in front of the base name.

If each local convolution/correlation uses the same UniKernel, "Uni" is appended after the base name. Otherwise, a field of kernels is expected, such that each local unit has its own filter / receptive field.

If the convolution / correlation wraps around at the boundaries, ie., the field is actually a two-dimensional torus, "cyclic" is appended to the name of the function.

Here is the full list of possibilities.

```
Matrix Correlate_2d ( Matrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix Correlate_2d_cyclic ( Matrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bCorrelate_2d ( bMatrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bCorrelate_2d_cyclic ( bMatrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )

Matrix Convolute_2d ( Matrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix Convolute_2d_cyclic ( Matrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bConvolute_2d ( bMatrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bConvolute_2d_cyclic( bMatrix in, Kernel kern, int x, int y,
                      int kx, int ky, Matrix out )

Matrix Correlate_2d_Uni ( Matrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix Correlate_2d_Uni_cyclic ( Matrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bCorrelate_2d_Uni ( bMatrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bCorrelate_2d_Uni_cyclic( bMatrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )

Matrix Convolute_2d_Uni ( Matrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix Convolute_2d_Uni_cyclic( Matrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
Matrix bConvolute_2d_Uni ( bMatrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
```

```
Matrix bConvolute_2d_Uni_cyclic ( bMatrix in, UniKernel kern, int x, int y,
                      int kx, int ky, Matrix out )
```

All these functions return "out", which must provide space for the results when a function is called.

Note that the cyclic functions are more time-consuming than the non-cyclic ones, and that UniKernels need less memory.

Later in this section another family of functions is introduced that extends the convolution/correlation functions to include lateral propagation delays, see section 4.6.

4.5.3 Orientation Tuning Maps

The following are a few functions that initialise single UniKernels or arrays of them (Kernels). They can be used to implement orientation tuning maps, but are rudimentary.

```
Set_Circ_Func_Uni_Kernel(UniKernel kern, int kx, int ky,
                BaseType (*func)(BaseType),
                BaseType height, BaseType width, BaseType offset)
```

Given a one-dimensional profile function "func" set a 2d-UniKernel "kern" to a circular symmetric profile. Kernel-dimensions are kx and ky. "height, width, and offset" set the amplitude and spatial scale, and an additive offset of the kernel, respectively. The function is additive.

```
void Gabor_Uni_Kernel ( UniKernel kern, int dimx, int dimy,
                        BaseType height, BaseType sigma1, BaseType sigma2,
                        BaseType kw, BaseType phikw, BaseType phisigmakw,
                        BaseType phi0 )
```

This function sets an UniKernel to have a Gabor-type receptive field, i.e., a 2d-sinusoidal wave modulated a by a spatial Gaussian function. "dimx" and "dimy" are the dimensions of the kernel. "height sets its amplitude. "sigma1" and "sigma2" are the standard deviations of the Gaussian along the first and second principal axes. "kw" is the wave-number. phikw is the orientation of the wave vector and phisigmakw the angle between the direction of the wave vector and the first principal axes of the Gaussian (usually believed to be 0 in cortical simple cells, but need not). "phi0" is the spatial phase of the sinusoidal. The function is additive.

```
void Set_Phi_Func_Kernel ( Kernel kern, int x, int y, int kx, int ky,
                           BaseType (*func)(BaseType),
                           Matrix phi,
                           BaseType height, BaseType width, BaseType offset)
```

This function takes a matrix of orientations, "phi" and generates a two-dimensional field of size x×y of two-dimensional kernels "kern" of size kx×ky. Each kernel has an orientation-tuned profile given by the scalar function "func" in a direction corresponding with the phi-value at the respective location in "phi". (Thus, these profiles can be plane waves, but can not in addition be Gaussian modulated as for Gabor wavelets. There is currently no dedicated function to set fields of Gabor wavelets at once.) "Height, width, and offset" have the same meaning as in the previous functions. The function is additive.

4.5.4 Layers and SpikeLayers

In order to make life easier in some applications, two types of fields have been defined with intrinsically stored sizes, Layers and SpikeLayers. These just redefine the more general structures described above, but use intrinsic variables xsize and ysize for their size.

```
#define SPIKE_LAYER ARRAY_CHAR_TYPE     // same as bMatrix
#define LAYER       ARRAY_FLOAT_TYPE    // same as Matrix

# define DEFAULTXSIZE  64
# define DEFAULTYSIZE  64

# define X_SIZE(_x) xsize = _x;
# define Y_SIZE(_y) ysize = _y;

extern int xsize, ysize;
```

Layers redefine Matrix and SpikeLayers bMatrix. Similarly Fields redefine Kernels and UniFields UniKernels. Default dimensions are 64×64, which can be changed using the macros X_SIZE and Y_SIZE above (in the function main_init()). Thereby, explicit size arguments can be often avoided:

```
Get_Layer()          // returns a Matrix of xsize * ysize
Get_SpikeLayer()     // returns a bMatrix of xsize * ysize
Get_Field(z,s)       // returns a field of xsize*ysize of kernels of size z*s
Get_UniField(z,s)    // returns a single kernels of size z*s

Free_Layer(l)
Free_SpikeLayer(l)
Free_Field(l)
Free_UniField(l)

Clear_Layer(l)
Clear_SpikeLayer(l)
Clear_Field(z,s,l)
Clear_UniField(z,s,l)

Fold_Spikes_Uni(inp, kern, kx, ky, out)
    same as:  bCorrelate_2d_Uni(inp, kern, xsize, ysize, kx, ky, out)

Fold_Spikes( in, kern, kx, ky, out)
    same as:  bCorrelate_2d( in, kern, xsize, ysize, kx, ky, out)
```

4.6 Delays

Delaylines are cyclic buffers that can store values of vectors and arrays of variables from previous steps. The user does not need to mess with the intrinsic data-structures of cyclic buffers. A number

of low-level access routines are provided as well as routines commonly encountered in dealing with delays in pool- and field-models.

4.6.1 Containers for Delay Variables

The following are types of container variables that can store different Felix types

```
Vector_DL
Matrix_DL
bVector_DL;
bMatrix_DL;
intVector_DL;
intMatrix_DL;
```

Allocating Delay Lines Use one of the following to allocate a delayline of a particular type. n, r, c are the number of elements, rows, columns, and l is the memory-length, ie, the maximum number of simulation steps that are stored.

```
Get_Vector_DL( _n, _l )
Get_Matrix_DL( _r, _c, _l )

Get_bVector_DL( _n, _l )
Get_bMatrix_DL( _r, _c, _l )

Get_intVector_DL( _n, _l )
Get_intMatrix_DL( _r, _c, _l )
```

Freeing Delaylines. Delaylines should be freed if no longer used by calling one of

```
Free_DL( _d )
Free_Vector_DL( _d )
Free_Matrix_DL( _d )
Free_intVector_DL( _d )
Free_intMatrix_DL( _d )
Free_bVector_DL( _d )
Free_bMatrix_DL( _d )
```

Note: calling just `free(dl);` is *not* enough. You need to use the above macros. It can be just `Free_DL( _d )`, however, to which all the other macros expand.

Resetting Delaylines. The following macros reset a delayline to a well-defined state; they do not clear the data buffers as such.

```
Clear_DL( _d )
Clear_Vector_DL( _d )
```

```
Clear_Matrix_DL( _d )
Clear_intVector_DL( _d )
Clear_intMatrix_DL( _d )
Clear_bVector_DL( _d )
Clear_bMatrix_DL( _d )
Clear_bitVector_DL( _d )
Clear_bitMatrix_DL( _d )
```

Setting Delaylines. The initial values of a delayline can be defined by a function "func" of parameters "P". This has to define values for each vector or matrix element and delay in the delayline. The calls below use the function to initialise a delayline.

```
void Set_Vector_DL( size_t n, size_t del, Delayline dl, float *P,
          BaseType (*func)(size_t x, size_t d, float *P) );
void Set_Matrix_DL( size_t rows, size_t cols, size_t del, Delayline dl, float *P,
          BaseType (*func)(size_t x, size_t y, size_t d, float *P) );

void Set_bVector_DL( size_t n, size_t del, Delayline dl,float *P,
          bBaseType (*func)(size_t x, size_t d, float *P) );
void Set_bMatrix_DL( size_t rows, size_t cols, size_t del, Delayline dl,float *P,
          bBaseType (*func)(size_t x, size_t y, size_t d, float *P) );

void Set_intVector_DL( size_t n, size_t del, Delayline dl,float *P,
          int (*func)(size_t x, size_t d, float *P) );
void Set_intMatrix_DL( size_t rows, size_t cols, size_t del, Delayline dl,float *P,
          int (*func)(size_t x, size_t y, size_t d, float *P) );
```

4.6.2 Accessing Containers

The following macros select delayed data-containers in a delayline "dl". It might be necessary to cast types in an application

current(dl): returns a pointer to the container for the current time-slice

last(dl): returns a pointer to the container for the previous time-slice

n_last(dl, n): returns a pointer to the container for the time-slice from "n" slices ago (it is not checked whether n is in proper bounds, ie $<$ memory length.

oldest(dl): returns a pointer to the container for the oldest time-slice (according to the memory length of the delay line

next(dl): returns a pointer to the container for the next time-slice

```
Step_DL(_d)
```

The macro `Step_DL` advances a delay line by one step (time-slice). It must be invoked after the updating of delayline data in the top-level step()-routine. It is assumed that step() stores newly

computed data in next(dl) (say, $x(t+h)$ for disretised differential equations or $x(t+1)$ for iterative maps). The routine increments the DL's current indexes and pointers; i.e recently computed data in "next" become "current".

4.6.3 Arbitrary Delays for Pools

Communication between two units in a network might take a certain time. In that case the connection is not only characterised by a number (synaptic strength), but in addition by a delay value. The subsequent two functions take delayed float or binary data "in" and multiply them by a coupling matrix "J", such that each individual connection has a delay as specified by the matrix "delays" (in simulation steps). The results are stored in the Matrix "out".

```
void  Mult_delayed_DL( int n,
                       Matrix J, int *delays,
                       Vector_DL in, Vector out);
void bMult_delayed_DL( int n,
                       Matrix J, int *delays,
                       bVector_DL in, Vector out);
```

Note: Delays are not checked for falling into range boundaries.

4.6.4 Convolution Functions with Distance-dependent Delays

In two-dimensional fields with local connectivities delays can be distance dependent according to some "axonal" propagation speed and possibly a fixed "synaptic transmittion" delay, too. The following functions generalise the convolution/correlation functions from section 4.5.2 to this case. Naming conventions are the same as there, but _delayed is appended to the function names in the case of finite lateral propagation. The input, of course, now must be a delay line of activities. Arguments "d" and "v" in the functions below are a fixed delay offset (synaptic delay) and the (axonal) propagation speed (in units / time step), respectively.

```
Matrix Convolute_2d_Uni_delayed( Matrix_DL in, UniKernel kern, int x, int y,
                       int kx, int ky, float d, float v, Matrix out);
Matrix Convolute_2d_Uni_cyclic_delayed( Matrix_DL in, UniKernel kern, int x, int y,
                       int kx, int ky, float d, float v, Matrix out);

Matrix bConvolute_2d_Uni_delayed( bMatrix_DL in, UniKernel kern, int x, int y,
                       int kx, int ky, float d, float v, Matrix out);
Matrix bConvolute_2d_Uni_cyclic_delayed( bMatrix_DL in, UniKernel kern, int x, int y,
                       int kx, int ky, float d, float v, Matrix out);

Matrix Convolute_2d_delayed( Matrix_DL in, Kernel kern, int x, int y,
                       int kx, int ky, float d, float v, Matrix out);
Matrix Convolute_2d_cyclic_delayed( Matrix_DL in, Kernel kern, int x, int y,
                       int kx, int ky, float d, float v, Matrix out);

Matrix bConvolute_2d_delayed( bMatrix_DL in, Kernel kern, int x, int y,
                       int kx, int ky, float d, float v, Matrix out);
```

```
Matrix bConvolute_2d_cyclic_delayed( bMatrix_DL in, Kernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);

Matrix Correlate_2d_Uni_delayed( Matrix_DL in, UniKernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);
Matrix Correlate_2d_Uni_cyclic_delayed( Matrix_DL in, UniKernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);

Matrix bCorrelate_2d_Uni_delayed( bMatrix_DL in, UniKernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);
Matrix bCorrelate_2d_Uni_cyclic_delayed( bMatrix_DL in, UniKernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);

Matrix Correlate_2d_delayed( Matrix_DL in, Kernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);
Matrix Correlate_2d_cyclic_delayed( Matrix_DL in, Kernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);

Matrix bCorrelate_2d_delayed( bMatrix_DL in, Kernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);
Matrix bCorrelate_2d_cyclic_delayed( bMatrix_DL in, Kernel kern, int x, int y,
                        int kx, int ky, float d, float v, Matrix out);
```

Note: It is not checked whether delays fall into proper bounds (must be smaller than the length of the delaylines). The possible axonal speed "v" and fixed additive delay "d" a thereby constrained.

Note further that theses functions are less efficient than their non-delayed counterparts. Cyclic boundaries cause an extra slow-down.

4.7 Random Numbers

Felix has an internal random number generator

Based on 4-state Mersenne twister ? x-check gsl ....

The Felix-intrinsic random number generator should be threadsave if OpenMP or MPI, or mixes thereof are used. However, because the parallel Felix extensions are quite recent, I haven't checked that intensively. (The binomial random number generator is known *not* to be threadsafe for $n >= 25$ and $n * p > 1$.)

Note also, that the initialisation in case of MPI/OpenMP parallel code is very simple. In order to have each thread generate a different sequence of random numbers, all threads contributing to a task are enumerated and the respective thread-numbers are just added to the seed provided to the randomize() function. This can lead to correlations in the numbers generated in different threads. I have no experience yet, how serious the effect can be. Send reports if you run into trouble caused by this overly simple procedure. (I'd then try using /dev/random which, however, is not very portable and has other disadvantageous).

`void randomize( int seed )` initialises the Felix intrinsic random number generator with "seed".

`long rand_long( void )` returns pseudo-random long integers in the range from 0 to $2^{32} - 1 =$

4294967295.

`float equal_noise( void )` returns equally distributed random numbers in the range $[0, 1.0[$.

`unsigned bool_noise( float p )` returns one with probability p and zero with probability $1-p$.

`float gauss_noise( void )` returns gaussian distributed random numbers with mean zero and standard-deviation 1.

`float lorentz_noise( void )` returns lorentz- (or cauchy-)distributed random numbers with mean 0 and standard-deviation 1.

`float binomial_noise( float p, int n )` returns binomially distributed random numbers $B(k; p, n)$ (as float values). [This generator is not threadsafe for $n >= 25$ and $np > 1$. It is mainly intended for implementations of synaptic failure, where n seems to be seldomly above 15 for cortical neuron types.]

Whereas, the previous funtions are all built on the same Felix-intrinsic random number generator, the follwing function (from Press et al) uses its own mechanism to generate random bits.

`unsigned int irbit( unsigned int * iseed )` generates a sequence of random bits, i.e., zeros and ones with equal probability. Iseed is some seed value. The sequences are not "very" random.

4.8 Sparse Vectors and Matrices

4.8.1 Sparse Vectors, semi-sparse Matrices

NOTE: Functions in this section might be subject to later changes as practicality considerations will indicate ....

Code for "sparse" vectors and matrices is currently being developed. Those appear to be useful in very large simulations where cells are only connected with a fraction of other cells. There is support for sparse floating point, binary (char), and integer vectors and matrices. The definitions for the floating point types are:

```
typedef struct
{
  int n,     // actual valid entries
      nmax;  // max entris befor reallocation
  int *i;    // indexes
  float *v;  // values
} sVector_t;

typedef sVector_t *sVector;


typedef struct
{
  int m;      // number of columns
  sVector *w; // array of column vectors
```

```
} sMatrix_t;

typedef sMatrix_t *sMatrix;
```

Binary and integer types have an additional 'b' or 'i' in their names, sbVector, siMatrix. These structures are actually "semi"-sparse only. sVectors are sparse, but sMatrices are sparse only in their rows; the array of columns is complete and not sparse. Each such sVector contains the sparse row-entries of that column. This reflects the fact that each neuron in a network projects to at least some other neurons. Similarly, each spike is distributed to at least *some* other cells.

4.8.2 Allocating, Loading, and Saving Sparse Arrays

The following functions corrspond with those for the standard Vector/Matrix types. Not all of these functions are fully implemented at the moment, in especially, none of the FILE I/O functions would work. The latter just print an error message at run-time, when called.

```
sVector Get_sVector( int size )
void Free_sVector( sVector v )
void Clear_sVector( sVector v )
void Empty_sVector( sVector v )
void Show_sVector( sVector v )
void Add_sVector_Entry( sVector, int i, float val )
float sVector_Elem( sVector v, int i)  // returns value v[i] or zero

void Write_sVector( sVector v, FILE*f )
void Read_sVector( sVector v, FILE*f )
void Save_sVector( sVector v, FILE*f )
void Load_sVector( sVector v, FILE*f )


sMatrix Get_sMatrix( int columns, int rows) // order matters
void Free_sMatrix( sMatrix w)
void Clear_sMatrix( sMatrix w)
void Empty_sMatrix( sMatrix w)
void Show_sMatrix( sMatrix w)
void Add_sMatrix_Entry( sMatrix w, int r, int c, float val )
float sMatrix_Elem( sMatrix w, int r, int c )

void Write_sMatrix( sMatrix w, FILE*f )
void Read_sMatrix( sMatrix w, FILE*f )
void Save_sMatrix( sMatrix w, FILE*f )
void Load_sMatrix( sMatrix w, FILE*f )
```

The same functions exist for binary and integer data types with an additional 'b' or 'i' in the names. Most of the function names should be self-explanatory. The difference between `Empty_sVector()` and `Clear_sVector()` ist that the first function just sets the number of active entries in the sVector to zero, whereas the 2cd function sets all active synapse to 0. The same holds for the sMatrix-equivalents.

`Add_sVector_Entry( sVector v, int i, float f)` adds an element with value "f" to an sVector at positions i. `Add_sMatrix_Entry( sMatrix m, int i, int j, float f)` does the same for position (i,j) of an sMatrix "m". If an sVector or sMatrix has to be increased in size, this should happen automatically. The floating point functions are additive – if the entry exists already, the new value is added to the old; for integers and binary data the old value is overwritten.

4.8.3 Sparse Matrix Vector Multiplications

The following are functions that multiply a sparse sMatrix with various other structures like Vectors, bVectors, sVectors, or integer arrays that just contain indexes of units supposed to be momentarily active.

```
Vector sMult, ( sMatrix w, Vector v, Vector out) );
Vector ssMult, ( sMatrix w, sVector v, Vector out ) );
Vector sbMult, ( sMatrix w, bVector v, Vector out ) );
Vector siMult, ( sMatrix w, int n, int *idx, Vector out ) );

Vector sMult_t, ( sMatrix w, Vector in, Vector out) );
Vector sbMult_t, ( sMatrix w, bVector in, Vector out) );
Vector sMult_t_delayed( sMatrix w, siMatrix d, Vector_DL in, Vector out )
Vector sbMult_t_delayed( sMatrix w, siMatrix d, bVector_DL in, Vector out )
```

The "xMult_t()"-functions do transposed multiplication, i.e., multiplication from the left; indexes in a column of a matrix are then interpreted as indexes of units where the respective cells *receive* input from. The xMult-functions in contrast assume that the columns contain *outgoing* synapses of a cell. It should (better) not be assume that any dimensions or arguments are checked. The extra argument in the delayed functions is a sparse matrix of integer valued delays of the same size as the weight matrix. It indicates which entries in the delay line "in" are relevant for a specific synapse.

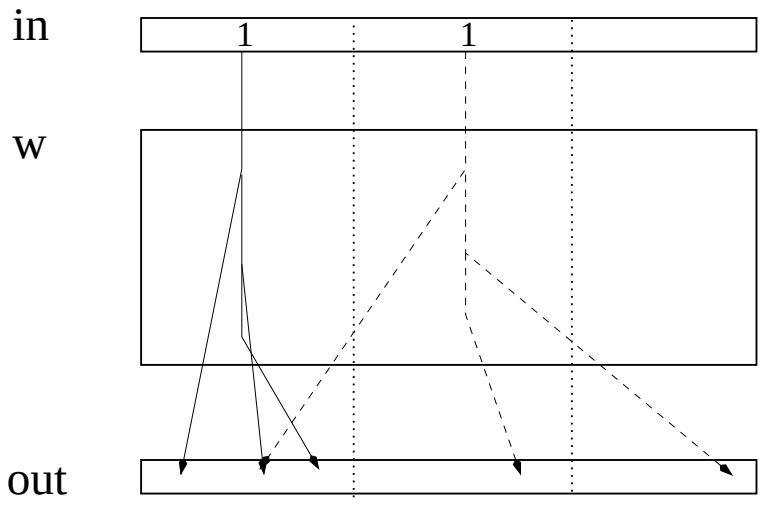

Figure 4.1: Scheme of multiplication of a sparse matrix and a binary Vector.

Figure 4.1 depicts sparse multiplication of a sparse matrix and a binary Vector. An outer loop would run over the input vector. Spikes (1's) in the input array can be distributed in a feedforward

way through the matrix, which contains all target indexes and weights. The weights are added to the respective entries in the target vector "out". This, however, can cross thread boundaries (indicated by dashed vertical lines), meaning that the same memory locations are potentially updated by different threads. This can not immediately be parallelised using OpenMP.

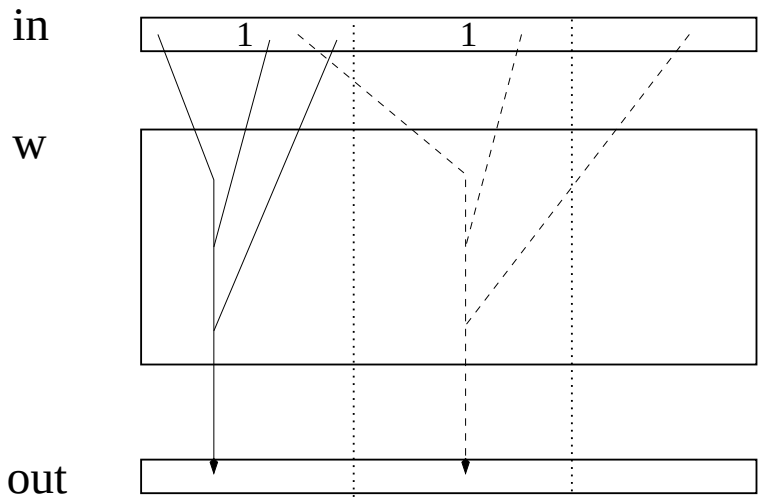

Figure 4.2: Scheme of transposed multiplication of a sparse matrix and a binary Vector.

Transposed multiplication solves this problem as shown in Fig. 4.2. Here the columns in a sparse matrix are interpreted as containing the indexes and weights of "incoming" synapses to units in the target vector "out". The outer loop then can run over the outputs, in which case each OpenMP-process would update a unique range of entries in the vector "out". Reading from the same location in different threads is not an issue. Even if running on several threads the routine can be less efficient as the previous one on a single thread. This is because it cannot make use of sparseness in the input vector as efficient as the forward multiplication.

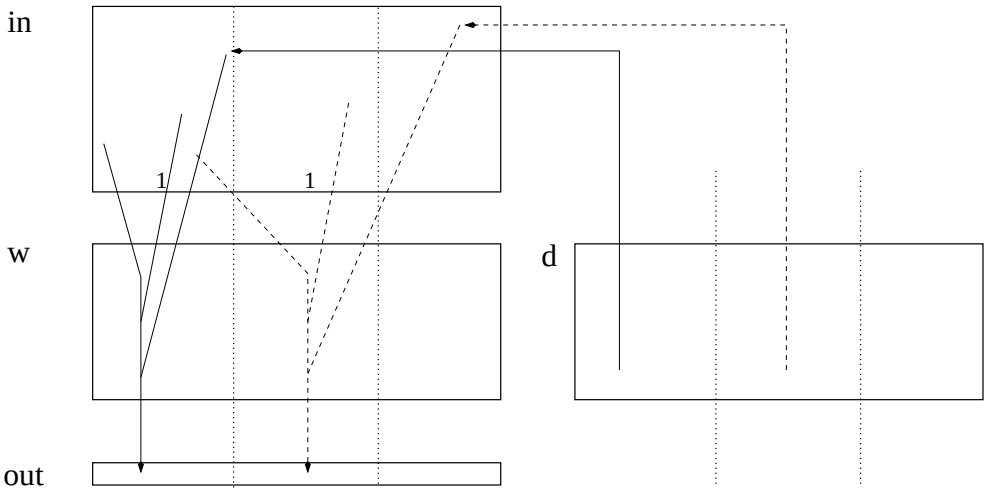

Figure 4.3: Scheme of transposed multiplication of a sparse matrix and a binary Vector with propagation delays.

Figure 4.3 displays how weights and delays interact in delayed sparse multiplication functions. The sparse delay matrix must have the same dimensions and represent the same connections as the

weight matrix. Whereas "w" provides the weights of synapses, the delay matrix determines which element in the input delay line has to be selected. OpenMP parallelisation is again easily possible (and implemented internally).

4.8.4 Orientation Tuning Maps with Distance-dependent Delays

```
sMatrix sCreate_Long_Range_Connectivities(
          int n, Vector in, float scale, float p, float theta );
```

This function takes a feature map "in" of size n and generates a sparse long range connection matrix based on pi-cyclic differences in the features. Synapses are not created if the differences in features are bigger than "theta" (in [0,1[where 1 means 'identical'). "scale" is an amplitude factor that sets the global scale (applied AFTER "theta"). "p" is an additional probability for creating synapses. Values in the feature map must be in the range [0...PI]. Autapses are not generated;

Note: This function can be used for 1d and 2d-feature maps. 2d-arrays "in" are reinterpreted as one-dimensional arrays of total size "n". In the 2d-case, however, co-linearity or other "Gestaltprinciples" (beside parallelism) are not taken into account.

```
siMatrix Make_Delays_from_Weight_Matrix( sMatrix w, int xsize, float d0, float v0 );
```

This function takes a weight matrix generated by the previous function and computes a delay matrix from it assuming a 1- or 2-dimensional network topology and distance dependent propagation speeds. If xsize is 0 a one-dimensional topology is assumed, otherwise, "xsize" is the size of the x-dimension in a 2D neural field (the number of columns, ie., total number of units, in the matrix must be a multiple of xsize in that case). d0 is a fixed delay and v0 the propagation delay. Units are in simulation time-steps and lateral units per simulation time respectively. The returned delays will be integers such taht they can be immediately used for indexing elements in a delay line.

The weight and delay matrices returned by the previous two functions can be used in conjunction with the sMult_t_delayed() and sbMult_t_delayed() functions.

4.8.5 Displaying Sparse Arrays in the GUI

The graphical user interface of Felix cannot display sparse Vectors and Matrix. You need to convert them befor, using, e.g.,

```
extern Vector Make_Vector_From_sVector( sVector v, int n, Vector out );
extern Matrix Make_Matrix_From_sMatrix( sMatrix m, int r, int  c, Matrix out );
```

"out" must point to memory space of appropriate space when these functions are called. A pointer to "out" is returned. "out" can then be used as usual as an argument to views in the graphical user interface.

4.8.6 Example: Sparse Integrate-and-Fire Network

Here is an example for a leaky-integrate-and-fire network with sparse connectivity. Only ten synpases per neuron/column are allocated from scratch. About a tenth per column are initialised by Gaussian random numbers. Missing synapses are automatically allocated.

Note that the system size is 900, because for small sizes the GUI takes most of the computation time (as long as display windows are open), which is unwanted for proper comparisons. In order to keep display windows at reasonable sizes, we have restricted the maximal sizes in the view declarations.

```
/* Example-program: sinf.c -- integrate and fire network
                            with sparse connectivity matrix */

# include <felix.h>
# include <sparse.h>

# define N   900    /* number of neurons      */
# define tau  10.   /* membrane time constant */

Vector  x;          /* potentials             */
bVector z;          /* vector of spikes       */
Vector  v;          /* auxiliary variable     */

sMatrix spJ;        /* sparse connectivity matrix   <<------------- */
Matrix  J;          /* connections for display */

SliderValue sI     = 100;  /* Common input to units  */
SliderValue sJ0    = 50;   /* Coupling strength      */
SliderValue ssigma = 0;    /* noise level            */

BEGIN_DISPLAY

 SLIDER( "input",     sI, 0, 200)
 SLIDER( "coupling", sJ0, 0, 200)
 SLIDER( "noise", ssigma, 0, 100)

 WINDOW("time courses")

  IMAGE( "x", AR, AC, x, VECTOR,  10, 10, 0.0, 1.0, 4)
  RASTER( "x", NR, AC, x, VECTOR, MIN(100, N), 0, 0.0, 1.0, 1)
  GRAPH( "x", NR, AC, x, VECTOR,  MIN(100, N), 0, 0, 0, -.01, 1.01 )
  RASTER( "out", NR, AC, z, bVECTOR, MIN(100, N), 0, -.01, 1.01, 2)

 WINDOW("couplings")

  IMAGE( "J", AR, AC, J, CONSTANT MATRIX,
                 MIN( 100, N), MIN(100, N), -4./N, 4./N, 2)
```

```
END_DISPLAY


NO_OUTPUT

int main_init()
{
  randomize( time(NULL) );
  SET_STEPSIZE( .1 )

  spJ = Get_sMatrix( N, 10 ); // only ten synapses per neuron
                              // are allocated from scratch
  J = Get_Matrix( N, N );
  x = Get_Vector( N );
  z = Get_bVector( N );
  v = Get_Vector( N );
}

int init()
{
  int i,j;

  Clear_bVector(N,z);
  Clear_Vector(N,v);

  for (i=0; i<N; i++)
    x[i] = equal_noise();

  Empty_sMatrix(spJ);                                        //  <<-----
  for (i=0; i<N; i++)                                        //  <<-----
    for (j=0; j<N/10; j++)  // only N/10 trials per column    //  <<-----
         Add_sMatrix_Entry( spJ, i , (int)(N*equal_noise()) , //  <<-----
                  10.0 / N * ( 1. + .4*gauss_noise() ) );     //  <<-----

  Make_Matrix_From_sMatrix( spJ, N, N, J ); // make a Matrix for the GUI
}

int step()
{
  int i;

  for (i=0;i<N;i++)
    leaky_integrate ( tau, x[i],
                  0.01*( sI + sJ0*v[i] + ssigma*gauss_noise() ) );

  Fire_Reset( N, x, 1.0, 0.0, z );  // firing and reset

  Clear_Vector( N, v );  // need to clear explicitly               // <<-----
  sbMult( spJ, z, v );   // sparse Matrix times non-sparse bVector // <<-----
```

```
}
```

4.9 Dynamic Synapses

In most technical neural networks synapses are simply represented by numbers, their weights. This is enough for many algorithms in learning theory, pattern recognition, or associative learning and retrieval. Biological synapses are more complicated. They reveal dynamic properties like facilitation and depression, they fail stochastically, and responses to stimuli do show transient time-courses usually characterised by so-called alpha-functions. It is possible to implement such properties in a Felix program using just the constructs descibed so far. This would require explicit code for the desired dynamic properties. Because they are of quite some importance in computational neuroscience dynamic synapses are now supported by Felix in a more systematic manner. There are new classes "SynapseVectors" and "SynapseMatrices" that integrate typical properties of biological synapses into the Felix core functionality.

Note: During the course of developing these components, the implemented functional properties and the underlying code got progressively more complex. Usage of SynapseMatrices is now a little more complicated as initially envisaged; the Felix-intrinsic code base is also not the most elegant. Later changes to this part of Felix are therefore not unlikely. For the time being, however, the synapses classes should be useable.

4.9.1 Types of Synaptic Dynamics

As explained, biological synapses are more than just numbers. They reveal a pretty rich variety of dynamical penomena. The most typical phenomena are:

Temporal response function: The response of a synapse to an incoming spike (e.g., in terms of transmitter release or post-synaptic changes in potentials) is a unimodal function of time, which rises with a certain time constant and decays roughly exponentially after having reached a single maximum. In the present document we call such functions alpha-functions. They can be described by "spike-response functions" which arise as responses of low-pass filters to short impulses (spikes) at the input. Depending on the number of subsequent low-pass filters the "order" of the alpha-function can be different. Common are 0, 1, and 2cd order alpha-functions, corresponding with jumps, decaying exponentials, and smoothly rising and falling post-synaptic potentials (or currents), respectively. More about this in subsecion **??**.

Adaptation and Facilitation: The total amplitude of synaptic responses can adapt on scales of typically several hundreds of milliseconds to the frequency of incoming spike-trains. Depending on whether the amplitude decreases or is suppressed one speaks of synaptic facilitation or adaptation.

Failure: Synaptic transmitter release is not a 100 percent reliable, but is a stochastic process. Embedded in the presynaptic membrane are discrete "vesicles" that contain roughly the same amount of neurotransmitter, the "release quantum". If a spike arrives at a synapse a certain small number of vesicles release their transmitter and can thereby evoke changes on the post-synaptic side. The number of released vesicles is well described by a binomially

destributed random variable where the probability of release at a single release site and the number of such sites can vary widely between synapse classes.

Models of these three phenomena have been described in the literature. Felix implements the most common of these models in a way that allows to combine their properties in any mixed synapse type.

Figure 4.4 displays a scheme of the generic synapse model. At a synapse, spikes are first fed into a failure stage according to the npq-model, then into a Barak/Tsodyks-stage for facilitation and depression, finally into a $0/1/2$-order low-pass filter that generates alpha-function-type conductance changes, $g(t)$. Each of the stages can be by-passed (not shown).

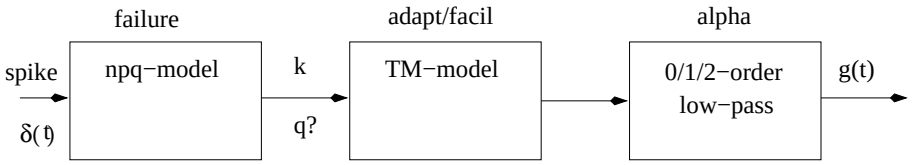

Figure 4.4: Three stages contributing to synaptic dynamics

Arrival of delta-spikes leads to generation of a binomially distributed randmon number k in the first stage. "Something" (see below) is fed into the BT-model in turn. That model has two variables u and x and three parameters: baseline U, and adaptation and facilitation time-constants τ_A and τ_F. Output of the BT-model is low-pass filtered to obtain alpha-function-type post-synpatic conductance changes.

The main problem when integrating the individual models occurs between the npq-model and the BT-model: What does the npq-model feed into the BT-model? $\delta(t)$, k*$\delta(t)$, kq*$\delta(t)$, kq/n*$\delta(t)$,...? How, in turn, impacts the "release probability" $u(t)$ of the BT-model on that of the npq-model, p?

The following sections describe the dynamics of the respective stages in more detail. Afterwards their combination is dealt with.

4.9.2 npq-model: synaptic failure

npq-model: A spike arriving at a synapse is assumed to release transmitter at a binomially distributed number of release sites out of a number of n. Release probability for a single site is p, and release quantum is q. So, after passing this stage we know the number of released sites, ie. a random variable, called k, the amount of released transmitter kq, the average npq, the fraction of sites that released k/n, etc.

4.9.3 BT-model: facilitation and depression

The Barak/Tsodyks-model as given in [?] for a single synapse reads

$$\frac{du}{dt} = \frac{U-u}{\tau_f} + U(1-u)s(t) \tag{4.3}$$

$$\frac{dx}{dt} = \frac{1-x}{\tau_r} - uxs(t) \tag{4.4}$$

The $s(t)$ are sequences of arriving Dirac-spikes. $U \approx .05, \tau_f, \tau_r$ are parameters. The amplitude of the evoked event is proportional to $u * x$. According to Barak&Tsodyks

U is the "utilisation", "analogous to release probability"

τ_f, τ_r are time-constants for facilitation and depression

$u(t)$ is the running value of utilisation; it is facilitated by every spike; decays to U with time-constant τ_f

$x(t)$ is the running fraction of available neurotransmitter in proportion to u; recovers to baseline 1 with time-constant τ_r

In response to a delta-spike the model reveals a typical amplitude of level U (up to facilitation and depression). If faciliation or depression are large, typically at high rates, the amplitude level can significantly deviate from U. This is, of course, desired.

Event-driven integration. As long as one is not interested in the precise time-course of u and x it is possible to use event-based simulation for the adaptation/facilitation-process, meaning that the variables u and x need only be updated at times where spikes arrive at a particular synapse, and not in every simulation time-step.

Figure 4.5 displays the time-course of u and x between two spikes at times t_n and t_{n+1}. Note that both variables are confined to the interval $[0, 1]$, which makes sense because u is "utilisation/release probability" and x is the "available fraction" of u. In fact, u is even always larger than U, the baseline level of u that is asymptotically reached if no spikes arrive for times $\gg \tau_F$.

According to the figure, the event-driven update at time t is (with $t = t_{n+1} - t_n$):

$$u(t_{n+1}-) = (u(t_n+) - U)\exp(-t/\tau_F) + U \quad (4.5)$$

$$u(t_{n+1}+) = u(t_{n+1}-) + \Delta u = u(t_{n+1}-) + U(1 - u(t?)) \quad (4.6)$$

$$x(t_{n+1}-) = 1 - (1 - x(t_n+))\exp(-t/\tau_F) \quad (4.7)$$

$$x(t_{n+1}+) = x(t_{n+1}-) - \Delta x = x(t_{n+1}-) - x(t?)u(t?) \quad (4.8)$$

In (4.5) to (4.8), $f(t_\pm) = \lim_{\epsilon\to 0} f(t \pm \epsilon)$, ie, the values of the function f immediately before $(t-)$ or after $(t+)$ time t.

Note the question marks in (4.6) and (4.8). At the time of spikes the variables u and x jump discontinuously, $u(t-) \neq u(t+)$ and $x(t-) \neq x(t+)$. It would appear natural to use the left-limits right before the spike arrives, however, it seems that in some of Tsodyks' papers the right-limit is used at least for $u(t?)$ in (4.8). This would mean that faciliation is practially instantaneous and adaptation slighly slower so that it depends on the already faciliated new utilisation value. This might or not be so. From a modellers point of few it is a matter of choice (multiplying the number of possible model variants by 2).

4.9.4 Alpha function conductance changes

There is not much to say about this third synaptic dynamics step. The output of the combined npq-BT-model is still a series of delta-functions, but of variable mass (depending on how many sites release transmitter, how high the facilitation level is, etc).

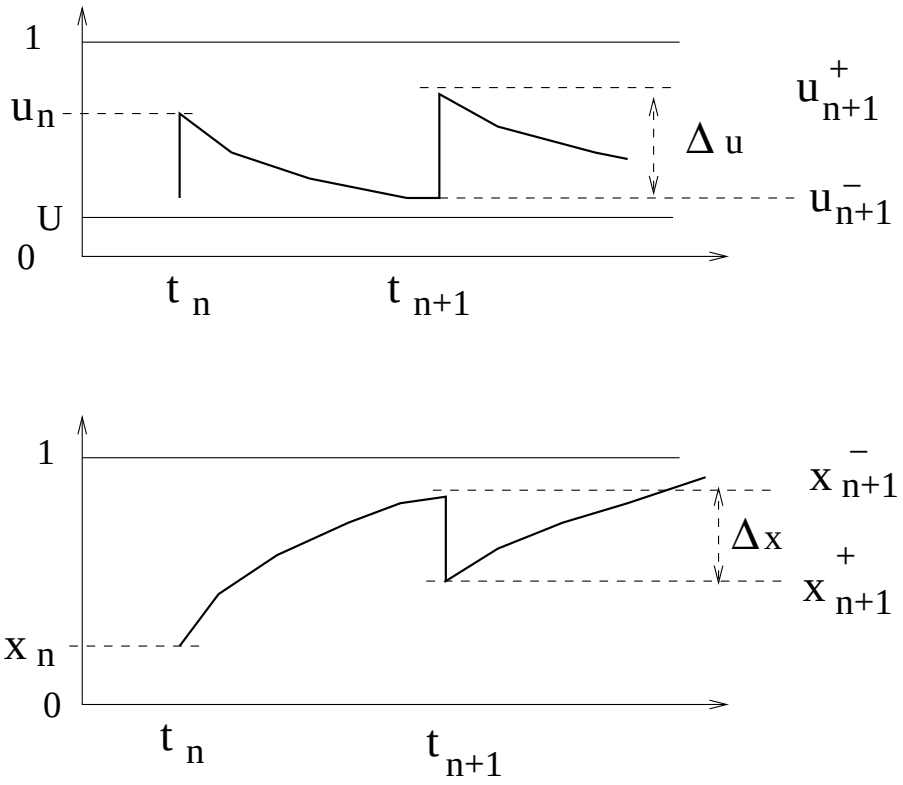

Figure 4.5: Time-course of utilisation $u(t)$ (release prob) and running (available) fraction thereof, $x(t)$, between two spikes.

In order to generate conductance changes, $g(t)$, the output is fed into a $0/1/2$-order low-pass filter (depending on choice). The resulting alpha-functions can then be used in dynamic equations for conductance based (4.9) or current based (4.10) membranes.

$$C_i \frac{dV_i}{dt} = -g_L \cdot (V_i(t) - V_L) + \sum_j g_{ij}(t) * (V_i - V_{rev}) \quad \text{conductance-based} \tag{4.9}$$

$$C_i \frac{dV_i}{dt} = -g_L \cdot (V_i(t) - V_L) + \sum_j g_{ij}(t) * (\bar{V}_i - V_{rev}) \quad \text{current-based} \tag{4.10}$$

V_{rev} is the reversal potential of the class of synapses and $\bar{V}_i$ the mean membrane potential of neuron i.

Note: In (4.10) $\bar{V}_i - V_{rev}$ is a constant in contrast to $V_i - V_{rev}$ in (4.9) (see Brette et al., [?]). This means (4.9) is more difficult to integrate in an event driven manner than (4.10). For some cases there are event-based schemes for (4.9) but (probably) no existing simulation tool implements them (depends on whether an explicit solution of the impulse response function is available or not; can be very tricky in general, see Brette 2006 [?], for a comparably "simple" case).

4.9.5 Coupling of npq- and BT-model

p in the npq-model is considered a release probability but u in the BT-model is, too. So, do they actually have to do something with each other? In the coupled npq-BT model we identify U, the baseline value of $u(t)$ in the BT-model, with p, the parameter for the release-probability in the npq-model and to choose for p in an actual spike event the running value of u.

That means, if a spike arrives, first a binomial random number for the releasing sites is generated according to $B(k; n, u(t_n-))$. A facilitated synapse thus will have a higher running value of the release probability $u(t_n-)$.

In the BT-model, U is in turn replaced by p from the npq model, such that asymptotically at low firing rates (compared to the facilitation/depression time-constants) u approaches the value p (which then is used effectively in the npq-model). If spike-frequency increases, the effective release probability (now $u(t)$) adapts or facilitates, accordingly.

$$u(t_{n+1}-) = (u(t_n+) - p)\exp(-t/\tau_F) + p \tag{4.11}$$
$$x(t_{n+1}-) = 1 - (1 - x(t_n+))\exp(-t/\tau_F) \tag{4.12}$$

For the jumps at spike times we choose

$$u(t_{n+1}+) = u(t_{n+1}-) + \Delta u = u(t_{n+1}-) + c_F k/n(1 - u(t-)) \tag{4.13}$$
$$x(t_{n+1}+) = x(t_{n+1}-) - \Delta x = x(t_{n+1}-) - c_A x(t-)k/N \tag{4.14}$$

Observe that we have added factors $0 \leq c_A, c_F \leq 1$ that can be used to control the amount of adaptation.facilitation after each spike. The original BT-model uses $c_A = c_F = 1$.

Note further that $E[k/n] = u(t-)$ in the present framework. For low firing rates $E[k/n] = u(t-) \to p = U$, such that the updates converge to the BT-limits up to the stochasticity of transmitter release. The updates with k/n replaced by $E[k/n] = u(t-)$ can be seen as some kind of "mean-field" model where the actual stochasticity in the transmitter release is replaced by the means of the released transmitter.

In the Barak-Tsodyks model the response to a spike is $\sim xu$. The output of the combined model discussed here is a series of delta-functions at the same times as the input spikes. Their amplitudes are $x(t-)k/n$, because k/n is the relative number of releasing sites (note, $E[k/n] = u(t-)$) and $x(t-)$ is the fraction of (remaining) utilisation (fresh = 1), that is if x is smaller than 1 less transmitter than maximally possible is released.

The parameter q from the npq-part is ignored in the present model. It would be an additional factor applied to the actual outputs. However, the model implementation already contains synaptic weights, which can incorporate the q values. This somewhat reduces memory space-requirements and numerical complexity. U the baseline level from the BT-model part is also ignored because it is identified with p.

4.9.6 Type Selection and Parameter Structures

Felix implements the three dynamic mechanism described above in a combinable manner in SynapseVectors and SynapseMatrices. In order to specifiy the desired mix several

macros have been define that can be used in constructirs for synapse matrices and vectors (see, e.g., 4.9.7). SYNAPSE_TYPE_ALPHA, SYNAPSE_TYPE_ADAPTATION, and SYNAPSE_TYPE_FAILURE select individual dynamic mechanisms, the other macros defined below provide convenient shortcuts.

Synapse Types

```
# define SYNAPSE_TYPE_ALPHA 0x01
# define SYNAPSE_TYPE_ADAPTATION 0x02
# define SYNAPSE_TYPE_FAILURE 0x4

# define ALPHA_SYNAPSE SYNAPSE_TYPE_ALPHA
# define ADAPTING_SYNAPSE (SYNAPSE_TYPE_ALPHA | SYNAPSE_TYPE_ADAPTATION)
# define FAILING_ALPHA_SYNAPSE (SYNAPSE_TYPE_ALPHA | SYNAPSE_TYPE_FAILURE )
# define FAILING_ADAPTING_SYNAPSE (SYNAPSE_TYPE_ALPHA |
                                   SYNAPSE_TYPE_ADAPTATION |
                                   SYNAPSE_TYPE_FAILURE )
```

Synapse Parameters

Each synaptic dynamic mechanism can be described by a number of parameters. Structures have been defined to collect these data.

```
AlphaParameters alpha;
AdaptParameters adapt;
FailureParameters failure;
```

The parameter structures have to be allocated before use and should be freed afterwards. A given structure can be set by the corresponding Set_xxx_Parameters function, whereas a copy of it is return by one of the Dup_xxx_Parameters functions. Show_xxx_Parameters prints the values of a parameter structure to screen (mainly for debugging).

```
AlphaParameters   Get_Alpha_Parameters( float taur, float tauf )
AdaptParameters   Get_Adapt_Parameters( float U, float tauA, float tauF,
                                                  float cA, float cF )
FailureParameters Get_Failure_Parameters( int n, float p, float q )

Free_Alpha_Parameters( r )
Free_Adapt_Parameters( r )
Free_Failure_Parameters( r )

AlphaParameters Set_Alpha_Parameters( AlphaParameters r,
                            float taur, float tauf )
AdaptParameters Set_Adapt_Parameters( AdaptParameters s,
                            float U, float tauA, float tauF,
                            float cA, float cF )
```

```
FailureParameters Set_Failure_Parameters( FailureParameters s,
                              int n, float p, float q )

AlphaParameters Dup_Alpha_Parameters( AlphaParameters r )
AdaptParameters Dup_Adapt_Parameters( AdaptParameters  r )
FailureParameters Dup_Failure_Parameters( FailureParameters r )

void Show_Alpha_Parameters(AlphaParameters r )
void Show_Adapt_Parameters(AdaptParameters r )
void Show_Failure_Parameters(FailureParameters r )
```

Local and Shared Parameters

Parameters can be local or global with respect to SynapseVectors or SynapseMatrices. In the first case each synpase may have individual values, whereas in the second they are shared among all of them. The latter obviously requires less memory and also allows for slightly faster code.

Whether a SynapseVector or SynapseMatrix uses shared parameters depend on how it is constructed and cannot be changed afterwards. If parameters for any of the three synaptic dynamic mechanisms are supplied during creation of a vector or matrix that parameter is global. Otherwise specific parameter sets must be supplied when synapses are actually added to the matrix.

4.9.7 Synapse Vectors and Matrices

SynapseVectors and SynapseMatrices have sparse entries in very much the same way as sparse Vectors and Matrices described in section 4.8. They just add dynamic mechanisms intrinsically. That is, Vectors and entries in Matrix colums are sparse, but the number of matrix columns is not. Again, this is motivated by the fact that each neuron usually does have at least a few synapses or, conversely, each spike is distributed to at least some neurons in a network.

SynapseVectors

SynpaseVectors are sparse vectors. They have to be allocated before usage and should be freed afterwards.

```
SynapseVector Get_SynapseVector( int n, int flags,
                                 AlphaParameters alpha,
                                 AdaptParameters adapt,
                                 FailureParameters failure )
void Free_SynapseVector( SynapseVector )
```

In Get_SynapseVector, n, is the initial size that may change as more synapses get added. flags define the type of the synapse, ie, which dynamic mechanisms it comprises. The type macros from subsection 4.9.6 have to be used here. The remaining three arguments are parameter sets for each of the three synpatic dynamic mechanisms. If any of these is non-zero, that respective parameter set is shared among all synapses in the vector. Parameter sets provided later if synapses are

actually added are ignored in this case. Note also, that parameters (local or shared) are ignored, if the respective type is not specified in the flags-argument. See, section 4.9.9 for an example.

Synapses are added to a SynapseVector v using

```
void Add_SynapseVector_Entry( SynapseVector v, int i, float weight, int delta,
                              AlphaParameters alpha,
                              AdaptParameters adapt,
                              FailureParameters fail)
```

Here, i, *weight*, and *delta* are the index, weight and time-delay (in multiples of the simulation time-step) of the addded synapse. Duplicate indexes overwrite previous entries. If a SynapseVector has local parameters for any of the different dynamic mechanisms, these parameters must be provided as arguments at synapse creation. If parameter values are supplied but that parameter set has been made shared during Vector creation, the new values are ignored.

A couple of functions exist to manage SynapseVectors

```
void Empty_SynapseVector( SynapseVector )

void Show_SynapseVector( SynapseVector )
void Show_SynapseVector_Index( SynapseVector sv )

.. more to come ...
```

Empty_SynapseVector discards allocated structures, except global parameters. This can be used if repeated reinitialisation are desired in the top-level init()-routine, but a SynapseVector is declared in main_init (as it would usually be the case).

Show_SynapseVector and Show_SynapseVector_Index are basically for debugging.

SynapseMatrices

SynapseMatrices are non-sparse arrays of SynapseVectors similar as for sparse Vectors and Matrices. Their functionality parallels that of SynapseVectors. Most of the functions below work in the same way as their vector counterparts. See previous subsection for further explanations.

```
SynapseMatrix Get_SynapseMatrix( int m, int n, int flags,
                                 AlphaParameters alpha,
                                 AdaptParameters adapt,
                                 FailureParameters failure )
void Free_SynapseMatrix( SynapseMatrix )

void Empty_SynapseMatrix( SynapseMatrix )

void Add_SynapseMatrix_Entry( SynapseMatrix, int, int,
                              float value, int delay,
                              AlpahParameters,
```

```
                                AdaptParameters,
                                FailureParameters )

void Show_SynapseMatrix( SynapseMatrix )
void Show_SynapseMatrix_Index( SynapseMatrix sm )

Matrix Make_Matrix_From_SynapseMatrix( SynapseMatrix m,
                                       int r, int  c, Matrix out )

Vector Get_Weight_Sums( SynapseMatrix w, Vector out )
```

The function Make_Matrix_From_SynapseMatrix converts the sparse weights of a SynapseMatrix into a non-sparse standard Matrix. This is necessary to display a weight matrix in the graphical user interface.

`Get_Weight_Sums` calculates a vector *out* of the sums of the synapses in the rows of a SynapseMatrix w.

4.9.8 Synaptic Matrix-Vector Multiplication and Updates

Sparse Multiplications

There are a number of multiplication functions analogous to those for sparse matrices:

```
Vector Synapse_Mult_t( SynapseMatrix w, Vector in, Vector out )
Vector Synapse_bMult_t( SynapseMatrix w, bVector in, Vector out )
Vector Synapse_Mult_t_delayed( SynapseMatrix w, Vector_DL in, Vector out )
Vector Synapse_bMult_t_delayed( SynapseMatrix w, bVector_DL in, Vector out )
```

These functions only use the synaptic weights for a sparse matrics vector multiplication. The dynamic properties even if they are set are entirely ignored. They are mainly for testing. It will typically be better to use sparse matrices if dynamic properties are not needed, because the sparse matrix functions should be more efficient in regard of space and time requirements.

Dynamic Synapse Matrix Updates

The following functions do matrix-vector multiplications on sparse matrices and update the internal SynapseMatrix structures per time-step.

```
Vector Synapse_bMult_Update_t_alpha( SynapseMatrix w, bVector in, Vector out )
Vector Synapse_bMult_Update_t_alpha_delayed( SynapseMatrix w, bVector_DL in,
                                             Vector out )

Vector Synapse_bMult_Update_t_failure( SynapseMatrix w, bVector in, Vector out )
Vector Synapse_bMult_Update_t_failure_delayed( SynapseMatrix w, bVector_DL in,
                                               Vector out )
```

```
Vector Synapse_bMult_Update_t_adaptation( SynapseMatrix w, bVector in, Vector out )
Vector Synapse_bMult_Update_t_adaptation_delayed( SynapseMatrix w, bVector_DL in,
                                                  Vector out )

Vector Synapse_bMult_Update_t_adaptation( SynapseMatrix w, bVector in, Vector out )
Vector Synapse_bMult_Update_t_adaptation_delayed( SynapseMatrix w, bVector_DL in,
                                                  Vector out )

Vector Synapse_bMult_Update_t( SynapseMatrix w, bVector in, Vector out )
Vector Synapse_bMult_Update_t_delayed( SynapseMatrix w, bVector_DL in, Vector out )
```

w is the SynapseMatrix under consideration, *in* a binary input vector or delay line (of spikes), and *out* the output (of instantaneous synaptic conductances).

The time-scale used for the internal update is set by the SET_STEPSIZE macro, see **??**. The current simulation step or time is returned by SIM_STEP and SIM_TIME, respectively.

The Synapse_bMult_Update_t_xxx_delayed - versions of the functions use delays defined per synapse. The functions without the _delayed suffix ignores delays even if they have been defined.

Different versions have been implemented for different combinations of alpha, adaptation, and failure. This has significant speed advantages. The Synapse_bMult_Update_t function and its delayed counterpart are wrapper that combine the more specific functions (see below).

Synapse_bMult_Update_t_alpha only uses the alpha-part of a dynamic synapse. If adaptation or failure parameters are defined at matrix creation, they are completely ignored.

Synapse_bMult_Update_t_failure only uses the failure part and if given also the alpha-part of a dynamic synapse. If adaptation parameters are defined, they are completely ignored. Parameter q is ignored in the present implementation (should be joined into the synapse weight).

Synapse_bMult_Update_t_adaptation only uses the adaptation/facilitation part and if given also the alpha-part of a dynamic synapse. If failure parameters are defined, they are completely ignored. This is the standard Tsodyks-Markram model as decribed above.

Synapse_bMult_Update_t_Up combines the npq-model with the Barak-Tsodyks model in the manner as described above. If an alpha-part is also given it is considered in this update-function, too. Parameters q and U are ignored in this model variant.

The above functions with the exception of Synapse_bMult_Update_t_delayed and Synapse_bMult_Update_t don't use the Matrix-type flags for deciding which dynamic mechanisms are used, because the kind of update is explicitely specified. The user has to make sure that Matrices are created with types that fit the respective update functions. Tests are usually not done, which can result in core-dumps.

The functions Synapse_bMult_Update_t_delayed and Synapse_bMult_Update_t are wrappers that call the other functions based on the Matrix-type flags. E.g., for Synapse_bMult_Update_t (and analogously for Synapse_bMult_Update_t_delayed):

```
case 0:               // just multiplication
  return Synapse_bMult_t(  w,  in,  out );
```

```
case SYNAPSE_TYPE_ALPHA:
  return Synapse_bMult_Update_t_alpha( w, in, out );

case SYNAPSE_TYPE_FAILURE:
case SYNAPSE_TYPE_FAILURE|SYNAPSE_TYPE_ALPHA:
  return Synapse_bMult_Update_t_failure( w, in, out );

case SYNAPSE_TYPE_ADAPTATION:
case SYNAPSE_TYPE_ADAPTATION|SYNAPSE_TYPE_ALPHA:
  return Synapse_bMult_Update_t_adaptation( w, in, out );

case SYNAPSE_TYPE_FAILURE|SYNAPSE_TYPE_ADAPTATION:
case SYNAPSE_TYPE_FAILURE|SYNAPSE_TYPE_ADAPTATION|SYNAPSE_TYPE_ALPHA:
  return Synapse_bMult_Update_t_Up( w, in, out );
```

Note: All of the above functions not only do the multiply-accumulate to compute conductance changes, but also update intrinsic data structures. Therefore, for each synapse matrix they have to be called exactly once in a single simulation time-step.

Note 2: In principle the columns of a SynapseMatrix can have different types, because they are SynapseVectors. This would allow to have parameters shared columnwise (ie neuron-wise). The Update functions should take different column types into account. However, this is a completely untested feature. (To use it one has to use low-level macros and data-structures, see synapse.c/h.)

4.9.9 Example: Integrate-and-Fire Network with Dynamic Synapses

The example below implements a network of leaky-integrate-and-fire neurons with dynamic synapses. The code looks very similar to earlier examples. Therefore, some parts have been left out. The main differences are indicated by arrows. Note that the synaptic dynamics as such is hidden from the user in the Synapse_bMult_Update_t-function – the leaky-integration in the step-function is only for the membranes. The flags SYNAPSE_TYPE_ALPHA, SYNAPSE_TYPE_ADAPTATION, SYNAPSE_TYPE_FAILURE in the initialisation of the synaptic matrix control the type mix of the synapses.

In this example, all synapses have identical parameters, because these are supplied already globally at initialisation of the matrix in the Get_SynapseMatrix-call in main_init(). This cannot be changed later. If some parameters need to be different for different synapses the respective parameter constructor needs to be replaced by 0 in the Matrix definition. Instead, it has to be specified when synapses are actually added to the matrix in the init()-routine using Add_SynapseMatrix_Entry(). Note that any parameters are ignored, if the corresponding type is not selected in the matrix definition. The types selected in the SynapseMatrix allocation specifiy which steps in the scheme in figure 4.4 are executed and which not.

Finally note, that the synapses in the example shown below have no delays; interactions are instantaneous, because the respective delay arguments when synapses are added to the matrix synJ in the init-function are 0. There is an example `syn_inf_del` in the Felix expl-directory, that shows how the program `syn_inf` can be modified to allow for delays.

```
/* Example-program: syn_inf.c
         integrate and fire network with sparse connectivity
         matrix of dynamic synapses with failure, adaptation,
         depression, and 0/1/2-order alpha-functions
 */

# include <felix.h>

# define N    900    /* number of neurons      */
# define tau  10.   /* membrane time constant */

Vector  x;          /* potentials             */
bVector z;          /* vector of spikes       */
Vector  v;          /* auxiliary variable     */

SynapseMatrix synJ; /* synaptic connectivity matrix */   <------------
Matrix  J;          /* connections for displaying */

BaseType mean;


SliderValue sI     = 100;  /* Common input to units  */
SliderValue sJ0    = 50;   /* Coupling strength      */
SliderValue ssigma = 0;    /* noise level            */

BEGIN_DISPLAY

  // same as sinf.c ; not repeated here

END_DISPLAY

NO_OUTPUT


int main_init()
{
  randomize( time(NULL) );
  SET_STEPSIZE( .5 )

  // starts empty with N columns; all parameters global   <-----------
  synJ = Get_SynapseMatrix( N, 0,
            SYNAPSE_TYPE_ALPHA
          | SYNAPSE_TYPE_ADAPTATION
          | SYNAPSE_TYPE_FAILURE,
            Get_Alpha_Parameters(  3., 5. ),         // tau_r tau_f
            Get_Adapt_Parameters( .05, 100., 500. ), // U tau_rec tau_facil
            Get_Failure_Parameters( 5, .3, 1. ) );   // n p q

  J = Get_Matrix( N, N );
```

```
  x = Get_Vector( N );
  v = Get_Vector( N );
  z = Get_bVector( N );
}


int init()
{
  int i,j;
  SynapseVector sv;

  Clear_bVector(N,z);
  Clear_Vector(N,v);

  for (i=0; i<N; i++)
    x[i] = equal_noise();

  Empty_SynapseMatrix(synJ);              <---------------------
  for (i=0; i<N; i++)                     <---------------------
    for (j=0; j<(int)(0.02*N); j++)
      Add_SynapseMatrix_Entry( synJ, i, (int)((N-1)*equal_noise()) ,
                                    1., 0,   // weight 1, delay 0
                                   0, 0, 0 ); // no local parameters

  Make_Matrix_From_SynapseMatrix( synJ, N, N, J );  <--- for display
}


int step()
{
  int i;

  for (i=0;i<N;i++) // current-based noisy integrate and fire neurons
    leaky_integrate ( tau, x[i],
                      0.01*( sI + sJ0*v[i] + ssigma*gauss_noise() ) );

  Fire_Reset( N, x, 1.0, 0.0, z );

  Synapse_bMult_Update_t_Up( synJ, z, v );   <---------------------

  mean = Sum( N, v )/N;
}
```

4.9.10 Patchy Connectivities in SynapseMatrices

It is often desired that a Neuron receives input from cells in a certain region, e.g., interneurons often sample activity from cells in their surrounding only.

The following few functions help setting up such kind of local connectivities. They take a SynapseVector sv which is supposed to hold the synapses of a target neuron and receives input from source cells around location $(x0, y0)$ in a field of size $n \times m$. Observe, that if $(x0, y0)$ are the coordinates of the cell itself, inputs will be sampled from the immediate surrounding of the cell, but $(x0, y0)$ can also be a location distant from the target neuron. In any case only synapses up to at most a distance $dmax$ from $(x0, y0)$ are created.

In these functions weights can be distance dependent according to a user-defined function (including the Felix-intrinsic `const_func`). The different versions of the functions deal differently with delays and the possibility to generate synapses will certain probabilities only. There can be constant delays (the same for all synapses created) or distance-dependent delays generated according to a user-defined function. Dilution of a connecivity pathway can similarly be controled by a distance-dependent function.

```
void Synapse_Add_Circular_Patch( SynapseVector sv,
                                 float x0, float y0, int n, int m, float dmax,
                                 float ampl, float scale, float (*func) (float),
                                 float delay )

void Synapse_Add_Circular_Patch_Delayed( SynapseVector sv,
                                 float x0, float y0, int n, int m, float dmax,
                                 float ampl, float scale, float (*func) (float),
                                 float dfac, float (*delayfunc)(float) )

void Synapse_Add_Diluted_Circular_Patch( SynapseVector sv,
                                 float x0, float y0, int n, int m, float dmax,
                                 float ampl, float scale, float (*dilfunc) (float),
                                 float ampl2, float scale2, float (*func) (float),
                                 float delay )

void Synapse_Add_Diluted_Circular_Patch_Delayed( SynapseVector sv,
                                 float x0, float y0, int n, int m, float dmax,
                                 float ampl, float scale, float (*dilfunc) (float),
                                 float ampl2, float scale2, float (*func) (float),
                                 float dfac, float (*delayfunc)(float) ) );
```

`Synapse_Add_Circular_Patch` sets weights within a radius $dmax$ according to $ampl * func(scale * d)$; all delays are set to $delay$. $func$ is a user-specified function, e.g., $gaussian$ or $const_func$.

`Synapse_Add_Circular_Patch_Delayed` in contrast to the previous function, this one sets delays according to $dfac * delayfunc(d)$, where $delayfunc$ is a user-supplied function (cf., the axample in the next subsection).

`Synapse_Diluted_Add_Circular_Patch` and `Synapse_Add_Diluted_Circular_Patch_Delayed` in addition to the previous two functions allow to setup diluted connections. $dilfunc$ is a distance-dependent function that specifies the probability of a connection. The function can be scaled by $ampl$ in order to set the total probability level; the function argument can be scaled by $scale$ in order to stretch or compress the range of the target function into an appropriate range.

Note: All these functions assume global parameters for the synaptic dynamics as set when the respective SynapseVector or SynapseMatrix is allocated.

Note 2: If synaptic dynamics is not required and connectivity is not heavily diluted the UniKernels, Kernels, and the convolution functions described earlier might be more advantageous to implement the intended functionality. They should be faster and need less memory space for non-diluted connectivities. They may still be faster for moderately diluted connectivities (which would be implemented using Kernels by zero-entries) because of better memory alignment and less overhead.

4.9.11 Example for dense local connections

Below is a brief code snippet showing how to set up local lateral connections with distance-dependent delays and a decaying connection probability. The distance-dependent delays are set by a user-defined function `delay_from_distance()`. Note that this function could also add some jitter to the delays as desired. The code leaves out the main_init() and step() which could be similar to the example in section 4.9.9.

```
float delay_from_distance( float d )
{
  return( 0.1 + 4.*d );  // d0+v*d
}

...

init()
{
  int i;
  SynapseVector sv;

  Empty_SynapseMatrix(synJ);
  OMP_FOR (i=0; i<N; i++) // auto-parallelises
  {
    SynapseVector sv = SynapseMatrix_Column( synJ, i );

    Synapse_Add_Diluted_Circular_Patch_Delayed( sv, i%nn, i/nn, nn, nn, 16.,
          .5, .2, gaussian,           //  dilution according to gaussian
          1., .2, const_func,         //  weights will all be equal to 1.
          .5, delay_from_distance );  //  delays according to user function
  }

  ...
}
```

4.10 Synaptic Plasticity

Weights of biological synapses can change in dependence of pre and post-synaptic activity. This phenomenon is called synaptic plasticity and generally addumed to underly learning processes taking place on a cognitive level.

A classic idea bout plasticity is the co-called Hebbain learning rule, which states that neurons that fire together should wire together, that is, strengthen their mutual synaptic connections. This makes sense because if these neurons often fire together they likely code for features in the world that belong together like parts of an object. Early theories of brain function suggest that object representations can build up this way.

Many variants and extensions of Hebbian learning rules have been devised and studied. For instance, unlearning (or synaptic long-term depression) when neurons do not fire together, or correlation-based learning rules, which consider post and pre-synaptic deviations from mean firing rates for learning and not the spikes or rates themselfes. More recently temporal learning rules have become important as the brain seems to make use of them widely. These rules named "spike timing-dependent plasticity rules" (STDP) usually enhance a synapse when a post-synaptic spike appears after a pre-synaptic one and decrease it in the opposite case. The properties of such learning rules are currently an important research topic in neuroscience.

Feix supports the implementation of some synaptic plasticity mechanisms. This however is a feature under development. Future changes and extensions to the functionality described in this section are lilkely..

The synaptic plasticity functionality of Felix builds upon SynapseMatrices as described in th eprevious section. Plasticity is an additional features you can give these matrices.

4.10.1 Plasticity Rules

If you want to use Felix plasticity functions, you have to define a plasticity rule that describes how a weight is changed given pre- and post-synaptic spike times. Th C function prototype of such an update rule is

```
void some_synapse_training_function( int j, int i, float*w, float tpre, float tpost );
```

As apparent from this prototype, at the moment a learning rule can depend only on the pre- and post-synaptic indexes and spike-times, and the current value of the synapse itself. This excludes some learning rules proposed in the literature as, e.g., recent rules explored in research that consider triplets of spikes. Rules that depend on further cell specific variables like post-synaptic potentials or average firing rates and the like (e.g., the BCM or ABS rule) may be possible as these variables can be computed in a program and used in a locally defined update function.

The example "learning_rules.c" in the code directory of this user guide provides some examples. However, these are experimental and have only used for testing. All Felix code developement in the are of SynapseMatrices and learning rules should be considered in an experimental stage.

The training function is called in a simulation each time a pre- or post-synaptic spike arrives at a synapse. These function calls are hidden in the update functions described in subsection 4.10.2.

You set a training function (usually in main_init) using

```
Set_Synapse_Training_Function( function_name );
```

Here, "function_name" is your own training function or one of the Felix intrinsic functions.

The training function defaults to an empty function that does nothing and is called "synapse_train_func_empty".

Note that if you don't want a synaptic projection (ie a SynapseMatrix) to learn it is probably better to use the update functions described earlier in the previous section about short-term synaptic dynamics than the update functions form subsection 4.10.2 below with empty training-rules, because otherwise you slow down your simulation by running through many unnecessary updates. See subsection 4.10.5 for benchmarks of these two groups of update-functions. (If you don't want sunaptic dynamics either, it might even be better to use just sMatrices or Matrices.)

The following code snippet shows how a training function for spike timing dependent plasticity (STDP) could look like. This is actually the training function used in the benchmarks reported in subsection 4.10.5. Note that the parameters used are not supposed to be realistic. In particular the weight changes have been set to quite small values in order not to disturb the firing rates in teh benchmark simulations much. You will probably use your own function(s) with more appropriate parameters.

```
void synapse_train_func( int j, int i, float *weight, float posttime, float pretime )
{
  float delta = posttime - pretime;
  float cp=.001, cm=.0003, taup=20., taum=50.;
  if (delta>0) // post after pre -> enhance
    *weight += cp*exp(-delta/taup);
  else // pre after post -> depress (but don't make negative)
  {
    if ( (*weight -= cm*exp(delta/taum)) < 0. )
      *weight = 0.f;
  }
  return;
}
```

At the moment synapse_train_func_empty and synapse_train_func are the only Felix-intrinsic training functions (but see example "learning_rules.c" in the code directory for more, experimental code).

4.10.2 Update Functions

Similar to the update functions for SynapseMatrices with dynamics synapses, there are a number of functions that update a SynapseMatrix and train the synapse simultaneously. These functions call the previously defined training function internally. There are versions for networks with and wthout delays. In the following function declarations *out* and *tout* are the postsynaptic spikes and last spike-times respectively, and *in* and *tin* are the input spikes and spike-times. In case of delayed functions the inputs have to be delay-lines of spikes.

```
void Synapse_Learn_t( SynapseMatrix w, bVector in, Vector tin,
                                       bVector out, Vector tout);

void Synapse_Learn_t_delayed( SynapseMatrix w, bVector_DL in,
```

```
                                    bVector out, Vector tout );
```

The above two functions leave the internal data-structures of w untouched, but only use the weights.

```
Vector Synapse_bMult_Learn_t( SynapseMatrix w, bVector in, Vector tin,
                              bVector out, Vector tout, Vector vout );

Vector Synapse_bMult_Learn_t_delayed( SynapseMatrix w, bVector_DL in,
                              bVector out, Vector tout, Vector vout );
```

These two functions also leave the internal data-structures of w untouched. In addition to training the weights they also compute the matrix-vector multiplication given the weights and input spikes. Resulting conductances (or currents depending on interpretration) are return in *vout*.

```
Vector Synapse_bMult_Update_Learn_t_adaptation( SynapseMatrix w, bVector in,
                              bVector out, Vector tout, Vector vout );

Vector Synapse_bMult_Update_Learn_t_adaptation_delayed( SynapseMatrix w,
                      bVector_DL in, bVector out, Vector tout, Vector vout );
```

These two functions do the same as the previous two, but in addition update internal data-structures of w, e.g., the alpha- and adaptation-variables.

```
Vector Synapse_bMult_Update_Learn_t_Up( SynapseMatrix w, bVector in,
                      bVector out, Vector tout, Vector vout );

Vector Synapse_bMult_Update_Learn_t_Up_delayed( SynapseMatrix w,
                     bVector_DL in, bVector out, Vector tout, Vector vout );
```

These two functions do the matrix-vector multiplication, train the synapses, and update the full Markram-Tsodyks equations with synaptic failure.

4.10.3 Unlearning

Classical associative memories like the Hopfield or Willshaw net store sets of binary patterns in synaptic connectivity matrices for later retrieval from incomplete or noisy versions of the patterns. Depending on whether both, only one, or none of the pre- and post-synaptic activity in a pattern are active a different increment can be added to a synapse when a pattern is presented. The increments can be collected in a rule-table, see Fig. 4.6. These networks learn before any simulations of the network dynamics are done by presenting all pattern pairs sequentially and changing synapses according to the rule-table. Furthermore, the networks are usually also time-discrete when retrieval is considered. It is therefore not entirely straight-forward to transfer local rule-tables to time-continuous networks with ongoing learning.

R	pre 0	pre 1
post 0	r00	r10
post 1	r01	r11

Figure 4.6: A local learning rule R adds increments $R_{post,pre}$ to a weight between synapses only based on the pre- and post-synaptic activity.

The code below shows an implementation of a Hebb-like learning rule. It uses the pre- and post-synaptic firing times together with a synchronisation inteval $[-t_{synch}, t_{synch}]$ in order to determine synchrony or cases where only the pre- or post-synaptic neuron has fired. According, weights can pe increased or decreased.

It is obvious that similar rules can be constructed that, e.g., take into account an exponential decay in increments in dependence of interspike intervals, thereby allowing for bigger increments if spikes are closer in time. Many other options are possible.

```
void synapse_train_func_hebb( int i, int j, float *weight, float posttime, float pretime
{
  float delta = posttime - pretime;
  float tsynch=10., r10=-.001, r01=-.001, r11=.003;

  if (delta>tsynch)      // post_not_pre -> r10
    *weight += r10;
  else if (delta<-synch) // pre_not_post
    *weight += r01;
  else
    *weight += r11;    // synch
}
```

One problem with the code above is that the case where both neurons *do not fire* cannot properly be detected. There is not event, no "spike", signaling this. Therefore, this case is excluded in the code snippet.

Often (but not always!) it is assumed that when a synapse does not receive any spikes for a long time it may "forget" the information it stores by some random perturbative processes acting on the weight. Such processes have been modeled by decaying synapses. The $r00$ term in a local learning rule might therefore be associated with synaptic "forgetting". This is not th emost general assumption, but a common one.

Whereas the synaptic training function is event based and only called if there is a pre- or post-synaptic spike at a synapse, one might guess that the no-pre-no-post case cannot be simulated event-based because it is not associated with an event. This is incorrect. Indeed, a synapses is updated at every pre- or post-synaptic event, so, if the next (pre- or post-synaptic) spike arrives, it is certain that the synapse has not been changed or even used during the time since the last update. We can therefore collect all the changes that would have happend according to the $r00$-parts of a

local learning rule and apply them just before the changes due to the new event.

For this purpose Felix provides the possibility to set up "forget_functions" which receive the pre- and post-synaptic neuron index, the current weight, and the absoulte time t of the last (pre- or post-synaptic) spike that led to any changes of the synapse.

```
Set_Synapse_Forget_Function( func )
void synapse_forget_func( int post, int pre, float* w, float t )
synapse_forget_func_empty
```

The function `Set_Synapse_Forget_Function` sets a forget-function. The default is 0 (equal to `synapse_forget_func_empty`). Below, an example function is defined that forgets the values of synapses that are not used on a long time-scale of 10000.0 (usually milliseconds).

```
void synapse_forget_func( int i, int j, float *w, float t)
{
  *w *= exp( (t-SIM_TIME)/10000. );
}
```

Note: Most of the update-functions with training in subsection 4.10.2 first call the forget-function, then do any adaptation, depression, failure, then determine the current weight of the synapse, and only after that train the synapse using the currently defined training-function. This reflects the fact that the forgetting happens during the time *before* the currently incoming spike, but the weight change tyically needs more time than the transient post-synaptic potential responses. (The Synapse_bMult_Learn_t and its delayed version are slightly different. Minor discrepancies to results from the other update-functions are possible.)

Note 2: ... last spike time problem ..... ouch .... in progress .... (in short: the actual binar spike vectors provided to an up-date function need to be from the current step, but the last spike times from the previous one in order to get the forget-functions ot work properly. So, compute the spikes, update the synapse structures, and then update the last spike time vectors at the end of your step function. See examples in learning_rules ....)

Note 3: there are 1 or 2 additional problems with the learning/forgetting I am still trying to figure out acceptable solutions for. Use them with care.

4.10.4 Example

The following code implements a network of roughly 4000 neurons in a square lattice of 63 times 63 units. Connectivity is 2%, e.g., each unit has up to about 80 synapses. This results in something less than 320k synapses. Synapses may or may not reveal delays, failure, depression/adaptation, alpha function dynamics, and synaptic plasticity according to th eSTDP-function in subsection 4.10.1. The step-routine contains function calls for a number of possible network update variants. All synaptic parameters are global for simplicity (but of course not their weightsm delays, and target indexes).

```
# include <felix.h>
```

```
# define CONNECTIVITY .02
# define nn 63
# define N (nn*nn) /* number of neurons 63*63=3969 */
# define tau 5. /* membrane time constant */

# define MAX_DELAY 150 // in time steps; make sure this is big enough

# define D 1 // display pixel size

Vector x; /* potentials */
Vector tl; /* last spike times */
bVector_DL zsav; /* output spikes buffer */
bVector z; /* pointer to actual spikes */
Vector v; /* auxiliary variable */

SynapseMatrix synJ; /* synaptic connectivity matrix */


SliderValue sI = 100; /* Common input to units */
SliderValue sJ0 = 50; /* Coupling strength */
SliderValue ssigma = 0; /* noise level */

BEGIN_DISPLAY

SLIDER( "input", sI, 0, 200)
SLIDER( "coupling", sJ0, 0, 200)
SLIDER( "noise", ssigma, 0, 100)

WINDOW("signals") IMAGE( "x", AR, AC, x, MATRIX, nn, nn, -.1, 1.1, D)
  IMAGE( "z", AR, NC, &z, POINTER TO bMATRIX, nn, nn, -.1, 1.1, D)

END_DISPLAY

NO_OUTPUT

NO_FMPI_CONNECTIONS

int main_init()
{

  OMP_THREADS( 1 );

  randomize( time(NULL) );
  SET_STEPSIZE( .5 )

  synJ = Get_SynapseMatrix( N, 0,
            SYNAPSE_TYPE_ALPHA | SYNAPSE_TYPE_ADAPTATION | SYNAPSE_TYPE_FAILURE ,
            Get_Alpha_Parameters( 3., 5. ), // tau_r tau_f
            Get_Adapt_Parameters( .05, 100., 700., 1., 1. ), // U tauA tauF cA cF
```

```
            Get_Failure_Parameters( 5, .1, 1. ) ); // n p q

  x = Get_Vector( N );
  tl = Get_Vector( N );
  v = Get_Vector( N );

  Set_Synapse_Forget_Function(0);               <------------ no "forgetting"
  Set_Synapse_Training_Function( synapse_train_func ); <-------------------

  zsav = Get_bVector_DL( N, MAX_DELAY );
  z = current( zsav );
}

int init()
{
  int i;
  SynapseVector sv;

  Clear_DL( zsav );
  z = current( zsav );

  Clear_Vector(N,v); for (i=0; i<N; i++)
  {
     x[i] = equal_noise();
     tl[i] = -1000.f;
  }

  Empty_SynapseMatrix(synJ);
  OMP_FOR (i=0; i<N; i++) // columns
  {
    int j, k;
    SynapseVector sv = SynapseMatrix_Column( synJ, i );

    for (j=0; j<(int)(CONNECTIVITY*N); j++)
    {
      k = (int)((N)*equal_noise()); // random source unit
      Add_SynapseVector_Entry( sv, k,
          1./(CONNECTIVITY*N), // synapse weight
         (int)delay_from_indexes( i, k, nn, 0., 2.), // dist. dep. delays
         0, 0, 0 ); // no local synaptic parameters
    }
  }
}

int step()
{
  int i;

  Step_DL( zsav );
```

```
  z = current( zsav );

  OMP_FOR (i=0;i<N;i++)
    leaky_integrate ( tau, x[i], 0.01*( sI + sJ0*v[i] + ssigma*gauss_noise() ) );
  Fire_Reset( N, x, 1.0, 0.0, z );

  // Synapse_bMult_t( synJ, z, v );
  // Synapse_Learn_t( synJ, z, tl, z, tl );
  // Synapse_bMult_Learn_t( synJ, z, tl, z, tl, v );
  // Synapse_bMult_t_delayed( synJ, zsav, v );
  // Synapse_Learn_t_delayed( synJ, zsav, z, tl );
  // Synapse_bMult_Learn_t_delayed( synJ, zsav, z, tl, v );
  // Synapse_bMult_Update_t_adaptation( synJ, z, v );
  // Synapse_bMult_Update_Learn_t_adaptation( synJ, z, z, tl, v );
  // Synapse_bMult_Update_t_adaptation_delayed( synJ, zsav, v );
  // Synapse_bMult_Update_Learn_t_adaptation_delayed( synJ, zsav, z, tl, v );
  // Synapse_bMult_Update_t_Up( synJ, z, v );
  // Synapse_bMult_Update_Learn_t_Up( synJ, z, z, tl, v );
  // Synapse_bMult_Update_t_Up_delayed( synJ, zsav, v );
  // Synapse_bMult_Update_Learn_t_Up_delayed( synJ, zsav, z, tl, v );

  for(i=0;i<N;i++) // update last spike times; not worth parallelising this
    if (z[i])
      tl[ i ] = SIM_TIME;
}
```

The above step-routine contains a number of commented out model variants. If none is selected the network consists of just ca 4000 unconnected leaky-integrate-and-fire neurons with gaussian noise input. Usually only one of the update functions will be active, with the exception of Synapse_bMult_t and Synapse_Learn_t (as well as their delayed counterparts). These functions complement each other. One updates the internal data structures for the synaptic dynamics, the other one does the synaptic plasticity. Note that Synapse_bMult_Learn_t (as its delayed version) integrate these two steps into a single more effcient function. It might however occasionally be useful to have the individual routines, too.

The source code of this example should be in docu/code/learn.c relative to the Felix main directory.

4.10.5 Some Benchmarks

Note: After doing the benchmarks reported here, some changes in the code have been done which slow down speed of some functions by up to about 20 %. You should also expect performance to depend to some degree on your particular compiler settings (ie, optimisation flags).

This subsection presents some benchmarking results for the training and updating functions defined above. The program used for these benchmarks is the one from the previous subsection with nn = 63, i.e., a total number of neurons of $nn * nn = 3969$. For an input of 1.01 and noise zero the firing rate of the uncoupled cells is ca 48Hz. Benchmarks are done with very small noise amplitudes, coupling strengths, and learning rates such that this baseline firing rate is not perturbed much but

the respective parts of the code are executed as desired. We simulated 1s real-time and the results shown below do contain the network setup phase and the actual simulation; or nn = 63 the setup phase however was short.

Benchmarks were run on a Laptop with Intel Pentium M processor 1.73GHz and a cache size of 2048 KB. The Felix version used was compiled with a custom-compiled pre-release of gcc 4.2 and run on one thread.

Dynamic parameters where the same for all synapses. If individual parameters are needed this would slow down the simulation. We have not run benchmarks for this situation.

Memory used for nn = 63 and a connectivity of 2% was 16MB most of which for the synapse intrinsic variables. nn = 63 corresponds with about 4k neurons in total. Given a connectivity of 2% each neuron had (up to) 80 synapses resulting in about 320k synapses in total. Each synapse needs about 32 bytes of memory for weights, delays, synapse indexes, last spike times, and the dynamic variables. The storage required for the synapses is therefore about 320k * 32 = 10.5MB. (A 100*100 network with 2% connectivity in contrast has 2M synapses and needs about 80MB on my Laptop. So, memory consumption is significant. Execution speed scales roughly with number of synapses.)

So, these are the numbers (in seconds). The alternatives correspond with those in the program code in subsection 4.10.4.

```
 0.8     leaky integration only
 2.5     leaky integration + gaussian noise
 1.3     leaky integration + firing&reset

 the following are all with leaky-integration, firing, and reset

 5.5     Synapse_bMult_t, no input noise
13       Synapse_bMult_t + Synapse_Learn_t, no input noise
15.5     Synapse_bMult_t + Synapse_Learn_t, gaussian input noise
10.5     Synapse_bMult_Learn_t, no input noise

11.      Synapse_bMult_t_delayed, no input noise
26.      Synapse_bMult_t_delayed + Synapse_Learn_t_delayed, no input noise
20.      Synapse_bMult_t_delayed + Synapse_Learn_t_delayed, gaussian input noise
22.      Synapse_bMult_Learn_t_delayed, no input noise
```

These values show that computing the single unit dynamics (leaky-integration, noise, firing&reset) is pretty much neglectable, and that the delayed functions are typically half as fast as the non-delayed ones, which is probably mainly due to unaligned memory access. Observe the speed benefit when the integrated update-learn functions are used. Gaussian input noise consistently costs about 1.5-3 seconds.

Here are numbers for the other update functions (all with leaky-integration, noise, firing&reset)

```
18.5  Synapse_bMult_Update_t_adaptation
24.5  Synapse_bMult_Update_Learn_t_adaptation
22.5  Synapse_bMult_Update_t_adaptation_delayed
30.   Synapse_bMult_Update_Learn_t_adaptation_delayed
```

```
  20.5  Synapse_bMult_Update_t_Up
  27.   Synapse_bMult_Update_Learn_t_Up
  22.5  Synapse_bMult_Update_t_Up_delayed
  33.5  Synapse_bMult_Update_Learn_t_Up_delayed
```

Delays cost about 4-7s as compared to non-delayed versions. Learning costs about 6-8s as compared to non-learning versions.

We have done preliminary test on the computer cluster. Networks were run on single compute nodes on either one 1 or 4 CPUs. For Synapse_bMult_Update_Learn_t_Up_delayed run-times were

```
            nn=63   100    200
1 thread:    40.7s  163s   61m8s
4 threads:   25.8s   63s   20m
```

For some reason still to be figured out simulation times are surprisingly bad on the cluster as compared to the Laptop. Although the CPUs on the cluster nodes have 2GHz cycle frequency (AMD Opterons) as compared to the 1.8 of the Laptop (Intel Pentium M) run-times on a single CPU are slower. The speedup on 4 CPUs is also rather low ($< 50\%$) but gets better for larger networks. The issue will be investigated further.

Memory consumption for the 200*200 network was 1.165GB on the cluster. That is 28.3% of the available 4GB. Memory consumption on the laptop were 16MB and 80MB for nn=63 and 100, respectively.

```
              units   synpase/unit          total synapses
  63*63    ~  4.000 * 0.02 = 80    * 4.000 =    320.000
 100*100   = 10.000 * 0.02 = 200   * 4.000 =  2.000.000
 200*200   = 40.000 * 0.02 = 800   * 4.000 = 32.000.000
```

Note that each synapse stores 7 integer/floating point numbers as intrinsic variables, synapse indexes, delays and weights resulting in 28Bytes if these numbers need 4 Bytes each. Givaen a network fo 200*200 units this results in 28B*32MB = 896MB for the synapses alone. This still assumes that all synapses shae their parameters, otherwise the local parameters (between 3 and 13 per synapse) have to be taken into account, too.

Non-dynamic synapse implemented by sparse sMatrices in contrast need only values for weights, indexes, and (possibly) delays resulting in 8 or 12 Bytes per synapse only. This allows for bigger networks. However, there are no training functions for sparse sMatrices yet.

4.11 Online Correlations

The computation of spike-triggered averages (STAs) and correlations is a common data-analysis method in neuroscience. Felix provides a couple of functions that compute STAs, cross- and auto-correlation functions online.

These functions use delay-lines to store previous data. The length of the delay-lines must be at least as big as the time window for the correlation functions to compute. The functions can compute several STAs of correlations functions at once. They expect vectors of data in the delay-lines and arrays of indexes that define which signal traces to use. The spike triggered averages in addition expect a vector of spikes for the triggers and an index array that specifies which triggers to use. The functions return arrays of STAs, CCFs or ACFs for all pairs of indexes.

The spiked-triggered averaging functions are

```
float*online_STA( int n1, bVector v,  // vector of triggers
                  int m1, int*indx1,  // index of triggers used
                  Vector_DL dl,       // data to average
                  int m2, int*indx2,  // index of datal channels used
                  int tau,            // max timestep used for STA
                  int flag,           // 0=one-sided; 1=two-sided STA
                  Vector out )        // results; m1*m2 array of STAs

int*online_bSTA( int n1, bVector v,  // vector of triggers
                 int m1, int*indx1,  // index of triggers used
                 bVector_DL dl,      // data to average
                 int m2, int*indx2,  // index of datal channels used
                 int tau,            // max timestep used for STA
                 int flag,           // 0=one-sided; 1=two-sided STA
                 int *out )          // results; m1*m2 array of STAs
```

The difference between both functions is that the first one averages floating point data, whereas the second uses chars - this can be binary 0/1 data but non-binary data is possible as well as long as they fit into chars. The second function uses integer arithmetics and therefore returns the STAs as integer arrays.

The functions compute $m1 * m2$ STAs at once. v is a vector of length $n1$; e.g., a vector of 0/1 spikes; these provide the "triggers" for the spike-triggered averaging. $indx1$ is an index vector of length $m1$; it selects relevant traces in v; other traces are ignored. dl is a delay-line of the data to average. The lengths of the vectors stored in the delayline needs to be bigger than any index appearing in $indx1$ and the number of stored vectors must be bigger than $tau + 1$ ($2tau+1$ if $flag$=1???), see below. $indx2$ is an index vector of length $m2$; it selects relevant traces in dl, other traces are ignored. The function computes the STAs for all selected triggers and data traces at once over a range defined by tau (in simulation steps, ie the temporal resolution of the data array). If $flag$ is non-zero, the average is computed over $2 * tau + 1$ steps symmetric in time around the current step, otherwise over $tau + 1$ previous steps. out is an array for the results or NULL. If NULL is provided an array of appropriate size is allocated. The address of out will be returned by the function

There are also cross- and auto-correlation functions. They use only a single delay-line and one index only. CCFs (ACFs) between (of) all seleteced traces are computed.

```
float* online_CCF( Vector_DL dl,      // data to correlate
                   int m, int*indx,   // index of datal channels used
                   int tau,           // max timestep used for CCF
                   int flag,          // 0=one-sided; 1=two-sided CCF
```

```
                  float*out )        // results; m1*m2 array of CCFs

int* online_CCH( bVector_DL dl,     // data to correlate
                 int m, int*indx,   // index of datal channels used
                 int tau,           // max timestep used for CCH
                 int flag,          // 0=one-sided; 1=two-sided CCH
                 int *out )         // results; m1*m2 array of CCHs

float* online_ACF( Vector_DL dl,    // data to correlate
                 int m, int*indx,   // index of datal channels used
                 int tau,           // max timestep used for ACF
                 int flag,          // 0=one-sided; 1=two-sided ACF
                 float *out )         // results; m1*m2 array of ACFs

int* online_ACH( bVector_DL dl,    // data to correlate
                 int m, int*indx,   // index of datal channels used
                 int tau,           // max timestep used for ACF
                 int flag,          // 0=one-sided; 1=two-sided ACF
                 int *out )         // results; m1*m2 array of ACFs
```

The file `tstacch.c` provides an example for the usage of some of the correlation functions.

4.12 numerics.c/h

This module contains a number of numerical support routines most of which have been adapted from example code coming with the excellent book by Press et al. [?]. Functions have been added as they became desired in the course of the author's research. In no way do they represent a comprehensive collection of numerical mathematics routines.

For detailed descriptions of the functions listed below have a look into Press et al.'s book.

4.12.1 Numerical Integration

Runke-Kutta of 4th order; and drivers with and without step-size control. See Press et al [?] for details.

```
void rk4( float*y, float*dydx, int n, float x, float h, float*yout,
              void (*derivs)(float, float *, float *) )

int rkdumb( float*vstart, int nvar, float x1, float x2, int nstep,
            void (*derivs)(float,float *,float *));

int rkqc( float*y, float*dydx, int n, float*x, float htry,
          float eps, float*yscal, float*hdid, float*hnext,
          void (*derivs)(float,float *,float *) );
```

```
int odeint( float*ystart, int nvar, float x1, float x2,
            float eps, float h1, float hmin, int*nok, int*nbad,
            void (*derivs)(float, float*, float*),
            int (*rkqc)(float*,float*,int,float*,float,
                        float,float*,float*,float*,
                        void (*derivs)(float,float *,float *) ) );
```

`rk4` computes a single Runge-Kutta step given a function `derivs` for the right hand-side of the diferential equation to integrate, $derivs(t, y, dydt)$.

`rkdumb` is a Runge-Kutta driver without step size control that does `nsteps` integration steps from `x1` to `x2` with initial values `vstart`.

`rkqc` is a Runge-Kutta driver with step size control. It does one step trying step size `htry` at an accuracy of `eps`. `yscal` provides relative weights of the scales of the variables. On exit `hdid` contains the possibly adapted step-size taken, and `hnext` suggest the next step size. `x` and `y` are updated to their new values.

`odeint` integrates a differential equation from `x1` to `x2` given `ystart` as initial values. `h1` is the initial step size and `hmin` a minimum steps size. `eps` specifies the accuracies of integration. `nok` and `nbad` contain the nuber of good and recomputed steps (with new step size) on exit.

4.12.2 Solving Matrix Equations

```
float Solve_Ax_b( int n, Matrix A, Vector b  );  /* b contains x on exit */
```

This function uses LR-decomposition, forward- and back-substitution. The function is destructive - a and b are overwritten. On exit b contains the result of Ax=b. A must be non-singular.

```
int gaussj(Matrix A, int n, Matrix B, int m)
```

Solves $Ax = b$ using Gauss-Jordan elimination with pivoting. `A` is an n $n \times n$ matrix, `B` an $n \times m$ matrix consisting of m right hand side vectors. On output, `A` is replaced by its inverse and `B` by the solution vectors. m can be zero, in which case `B` remains unchanged, and `A` is inverted. This function is not from Press et al. but rather from some original publication.

4.12.3 Eigenvalues

```
int Eigen_Values( int n, Matrix A, Vector wr, Vector wi)
```

Computes the eigenvalues of a real $n \times n$ matrix. Returns the real and imaginar parts in the arrays wr and wi, respectively. Uses balancing and Hessenberg form.

4.12.4 Nonlinear Least-Square Fitting

```
void mrqmin(float*x, float*y, float*sig, int ndata,
            float *a, int ma, int*lista,
            int mfit, float*covar, float*alpha,
            float*chisq, float*alamda,
            void (*funcs)(float,float *,float *,float *,int) )
```

Levenberg-Marquart method attempting to reduce the value chi-square of a fit between a set of points `x[0..ndata-1]`, `y[0..ndata-1]` with individual standard deviations `sig[0..ndata-1]` and a nonlinear function depending on coefficients `a[0..ma-1]`. The array `list[0..ma-1]` numbers the parameters such that the first `mfit` correspond to values actually being adjusted. the remaining parameters are held fixed at their input values.

The program returns current best fit values for the `ma` fit parameters, and chi-square. The `[0..mfit-1][0..mfit-1]` elements of the array `covar[0..ma-1][0..ma-1]`, `alpha[0..ma-1][0..ma-1]` are used as working space during most iterations.

Supply a routine `funcs(x, a, yfit, dyda, ma)` that evaluates the fitting function `yfit`, and its derivatives `dyda[0..ma-1]` with respect to the fitting parameters `a` at `x`. On the first call provide an initial guess for the parameters `a`, and set `alamda<0` for initialization (which then sets alamda=.001). If a step succeeds chisq becomes smaller and alamda decreases by a factor of 10. If a step fails alamda grows by a factor of 10. You must call this routine repeatedly until convergence is achieved. Then make one final call with alamda=0., so that covar returns the covariance matrix, and alpha the curvature matrix (and some allocated memory is freed).

```
void mrqdriver(float*x, float*y, float*sig, int ndata,
               float*a, int ma, float*covar, float*alpha, float chisq,
               void (*funcs)(float,float *,float *,float *,int) )
```

A driver for `mrqmin()` that assumes that all `ma` parameters a of `funcs` are fitted. On init chisq determines the chi square value which should be reached. The function exits if either the actual chi value falls below this initial chisq or if `MRQ_MAXITER=15` iterations are performed. On exit chisq contains the final chi square value.

4.12.5 Root Finding

```
float rtbis( float (*func)(float), float x1, float x2, float xacc)
```

Root finding by bisectioning, finds a root of `func` in the interval `[x1,x2]` with accuracy xacc; on entry func(x1)*func(x2) must be lower than 0.

```
NDiff( int n, float*y, void (*func)(int n, float*y, float*dy), float*dfy )
```

Compute partial derivatives dfy of function func past y. y and f must be n-dimensional; dfy an $n \times n$ matrix

```
int Solve_Fx( int n, float*y,
     void (*func)( /* n, y, f(y) */ ) ,
     void (*derivs)( /* n, y, dfdy(y) */ ) )
      /* if derivs==NULL: use numerical differentiation */
```

Solve a set of n nonlinear equations `func(y) == 0`, where y is n-dimensional, too. If available, `derivs()` should compute the matrix of partial derivatives. If derivs si NULL, derivatives are computed numerically. The function returns 0 on success; -1, if the matrix of derivatives gets singular, i.e., fixed point iteration is no longer possible, and -2 if the maximum number of iterations is reached. In case of success, y returns the solution vector.

4.12.6 Optimization

```
void mnbrak(float*ax, float*bx, float*cx,
           float*fa, float*fb, float*fc,
           float (*func)(float))
```

Given a function `func`, and given distinct initial points `ax` and `bx`, this routine searches in the downhill direction (defined by the function as evaluated at the initial points) and returns new points `ax`, `bx`, `cx`, which bracket a minimum of the function. Also returned are the function values at the three points, `fa`, `fb`, and `fc`.

```
float brent(float ax, float bx, float cx,
     float (*f)(float), float tol, float*xmin)
```

Given a function `f` and given a bracketing triplet of abscissas `ax`, `bx`, and `cx` (such that bx is between ax and cx, and f(bx) is less than both f(ax) and f(cx)), this routine isolates the minimum to a fractional precision of about `tol` using Brent's method. The abscissa of the minimum is returned as `xmin`, and the function value as brent, the returned function value.

Here is an example of how to use mbrak and brent:

```
float ax, bx, cx, fa, fb, fc, tol, xmin;

ax = .2; bx = .1;
mnbrak( &ax, &bx, &cx, &fa, &fb, &fc, sinf );
printf("x^2 :: ax = %f  bx = %f cx = %f  fa = %f  fb = %f  fc = %f\n",
          ax, bx, cx, fa, fb, fc );
tol=0.001;
fb = brent(ax, bx, cx, sinf, tol, &xmin);
printf("x^2 :: xmin = %f fmin = %f  tol = %f\n", xmin, fb, tol );
```

If the derivative of the function to minimize can be computed the following modification of `brent` is advantageous:

```
float dbrent(float ax, float bx, float cx,
            float (*f)(float), float (*df)(float),
            float tol, float *xmin)
```

Given a function f and it's derivative function df, and given a bracketing triplet of abscissas this routine isolates the minimum to a fractional precision of about tol using a modification of Brent's method that uses derivatives. The abscissa of the minimum is returned as xmin and the minimum value as dbret, the returned function value.

The following functing searhces for a minimum of an n-dimensional function if derivatives are not available.

```
void powell(float*p, float*xi, int n, float ftol,
            int *iter, float *fret, float (*func)(float *) )
```

Minimization of a function `func` of `n` variables. Input consists of an initial starting point `p`, and initial matrix `xi[][]` whose columns contain the initial set of directions (usually the `n` unit vectors), and `ftol`, the fractional tolerance in the function value such that failure to decrease by more than this amount on one iteration signals doneness. On output, `p` is set to the best point found, `xi` is the then-current direction set, `fret` is the returned function value at `p`, and `iter` is the number of itertions taken. The routine `linmin` is used.

Here is an example of hwo to use `powell`

```
float xsquare2( float *x )
{
  return x[0]*x[0] + x[1]*x[1];
}


...
  float p[2]={1.,1.}, xi[4]={ 0., 1., 1., 0.}, fret=0;
  int iter=0;
  powell( p, xi, 2, 0.001, &iter, &fret, xsquare2 );
  printf("(x,y) = (%f, %f); f = %f;  iter = %d\n", p[0], p[1], fret, iter );
```

If derivatives of the function to minimize are available use the following function for the minimization.

```
void frprmn(float*p, int n, float ftol, int*iter,
            float*fret, float (*func)(float *),
            void (*dfunc)(float *, float *))
```

Given a starting point `p`, Fletcher-Reeves-Polak-Ribiere minimization on a function `func`, using its gradient as calculated by routine `dfunc` is performed. The convergence tolerance on the function value is input as `ftol`. Returned quantities are `p` (the location of the minimum), `iter` (the number of iterations that were performed), and `fret` (the minimum value of the function).

Chapter 5

File I/O

The very basics of the file output functionality of Felix have been described in the quick-start chapter 2. We now look a little deeper into the possibilities.

Felix was used over the years mainly to either study autonomous dynamical systems and neural networks, or systems where stimuli could be computed as part of the simulation (e.g., simple bars and graitings). So far, there has never been much need for advanced file-input features and, therefore, Felix provides only some support for *output* of data to files. However, you can always use the standard C methods to load and store data from files (FILE objects, raw and formatted I/O, etc).

Even the file-output properties that are supported are not fully developed. Some facilities, which I imagined would be nice to have years ago, heve actually never been implemented, others never completed. What I describe below are features that I use often or have at least used occasionally.

5.1 Interface for File Output

The philosphy of the file-output interface is similar to that of the graphical display: One has to define a top-level function "MakeOutFiles()", which contains specifications of "OUTFILEs" (analog to "WINDOWs"), which in turn can comprise a variable number of "SAVE_VARIABLEs" (analog to "views on data" or graphics objects in the GUI).

The top-level MakeOutFiles()-routine can be either explicitely defind or constructed by using the macros

```
#define BEGIN_OUTPUT  void MakeOutFiles(){
#define END_OUTPUT    }
#define NO_OUTPUT     void MakeOutFiles(){}
```

Note that NO_OUTPUT expands into an empty function body. In that case no output will be written to external files through the interface mechanisms (but possibly through raw I/O, see section 5.3).

If output files are declared, a button will appear in the graphical user interface, see, Figure 2.4, that actually switches the output on or off during a simulation. The button label reflects the

current state. If the button is right-clicked some further control elements appear, which show the files defined, which variables they contain, and some elements that allow to change several file setting interactively. Be aware that not all of the functionality is fully implemented.

Beside using the GUI-Save-button, it is also possible to switch file I/O on or off from the source code by using the macros

```
SAVE_ON
SAVE_OFF
```

Another macro that often is useful influences the format of ASCII output. The macro

```
SET_ITEM_SEPARATOR( sep )
```

takes a string and inserts it between subsequent entries in the output. The macro should be placed right after the head of the the MakeOutFiles()-function. Default for the item separator is a single blank (`" "`), but this can cause problems with very long linelengths in files that have to be read from another program. Some tools for postprocessing (e.g., gnuplot) also expect only a single entry per line (by default in some modes), in which case the item separator can be set to newline (`"\n"`).

5.1.1 Output Files

Inside the function MakeOutFiles() one or more output files have to be defined using the macro

```
OUTFILE(name)
```

where "name" is the name of the file. If the file does not exist and data is written, it will be created, otherwise the old file will be overwritten.

The macro OUTFILE returns a file handle of type `int`. It is not often necessary to save the handle, but some of the later functions make use of it.

Output files are declared in serial order (as WINDOWs in the GUI). Instead of the file handle one can also use the macro `THISFILE`, which expands to the currently active file (ie the most recently declared one).

The file handles are only necessary if an application needs to set file-properties explicitly. The macros

```
FILE_ACTIVE( fileno )
FILE_INACTIVE( fileno )
```

anywhere in the code can, for instance, switch file-output to a particular file on or off. (This, however is further controled by the global SAVE_ON/SAVE_OFF switch. As long as that "master" switch is off, nothing will be saved.)

Other file properties are file format (raw (default) or ASCII)) and the behaviour in case the file is switched on and off more than once in a simulation (data can be overwritten or appended). These flags are set using

```
SET_SAVE_FILE_FLAG(fileno, flag, val )
```

where fileno is the file-handle (or "THISFILE"), flag is "ASCII" for declaration of the output mode and "APPEND" for the reset mode. Possible values for "val" in both cases are ON and OFF, i.e. `SET_SAVE_FILE_FLAG(THISFILE,ASCII,ON)` would switch ASCII output on for thelast recently decalred file in the MakeOutput()-function. (Note that it does not make sense to switch between both modes during one simulation. The files would then at least be relatively difficult to read; depending on the platform/C-implementation results can even be undefined).

The file flags should be set right after the declaration of an output file, ie, before any output variables.

If ASCII mode is on, an empty line will be saved after each vector or row of a matrix, and an extra newline after each complete matrix. The current step will also be saved on an individual line starting with the double-cross # befor all other data in that step. No such extras are saved in raw mode, just pure binary data.

5.1.2 Output Variables

Each output file can contain a number of output variables declared by the macro

```
SAVE_VARIABLE(name, var, type, dim_x, dim_y, flags, when, which)
```

Meaning of the argments is very similar to the various graphical views on data (see, section 3.3.2).

"name" is a string for the name the entry appears under in the graphical user interface.

"var", "type", "dim_x", and "dim_y" are the variable to store, its type, and dimensions. The types and specification of dimensions are the same as for graphics objects in display windows (MATRIX, VECTOR, etc.), see section 3.3.4. POINTER types are possible.

"flags" are output variable-specific flags that are mainly used to specify which data entries are stored when. Default is that each value is stored in each step (as long as the gobal save switch and the respective file-switch are ON)

"when" and "which" are further used to declare spatial and temporal selections of data to store in detail. This is useful in large simulations where output files can easily become very large. The options for sub-selections are explained in the subsequent two sections.

5.1.3 Temporal Selections

By default (and only if the save-button is activated) data is saved after a call to the top-level init()-function (to save "initial values") and after every simulation step. This can be modified individually for each SAVE_VARIABLE using the "flags and when" arguments in their declaration.

A CONSTANT variable that doesn't change during a simulation can be declared by an ONINIT flag. Such a variable is then only saved after calls to init(), because there is where it would naturally be initialised. Possible flags are:

```
ONINIT

SKIP
RANGE
SELECT
```

The last three flags correspond with three functions as arguments to the "when"-argument of the SAVE_VARIABLE declaration:

TSkip(skip) : Only every "skip" step is stored

TRange(start, stop, skip) : Data is stored at regular intervals starting at time step "start", storing every "skip" steps, up to a maximum step of "stop"

TSelect(n, vals) : "vals" is an integer array of size "n" that defines points in time when the data has to be saved.

A few examples are shown in subsection 5.1.6.

5.1.4 Spatial Selections

As in the temporal domain, selections can also be made spatially, more precisely, in one- or two-dimenional arrays. By default, *all* entries in an array-variable (MATRIX, VECTOR, etc.) are stored, if the temporal selection permits it. Alternative options are GRID, IRR_GRID, or POINTS, which refer to regular grids, irregular grids, and sets of individual points/coordinates, respectively.

As for the temporal selections the spatial selection (if it is not ALL) has to be notified in the flag-argument of a SAVE_VARIABLE (see above) using one of

```
GRID
IRR_GRID
POINTS
```

The precise selection has then to be specified as the final "which"-argument of a SAVE_VARIABLE declaration using one of the corresponding functions

Grid(start, stop, skip, start2, stop2, skip2) : This can be used for regular subgrids. "start, stop, and skip" are the first and maximal index of stored elements in the first dimension (x) and "skip" is the regular interval between indexes. The same meaning applies to "start2, stop2, and skip2" in the second dimension (y). For one-dimensional arrays start2, stop2, and skip2 should be zero.

Irregular(nx, values_x, ny, values_y) : This defines an irregular grid, where the integer array "values_x" contains "nx" coordinates in the first dimension and likewise for "ny, values_y". Data is saved for matrix entries at all pairs of x and y values. For one-dimensional arrays the ny and y-values should be zero.

Points(n, values_x, values_y) : This is the most general option because it allows for arbitrary coordinates in the index (integer) arrays "values_x, values_y" of size "n". Data values at the respective n points are saved. For one-dimensional arrays "values_y" should be zero.

A few examples are shown in subsection 5.1.6. Index boundaries are not checked. It is the programmers responsibility to make sure indexes do not exceed array-dimensions. Order for two-dimensional Grid() and Irregular() grid data is left-right (x first), then top-bottom (y).

5.1.5 The Timer

The timer (or Stop Watch) is a further facility to control when storage of data starts and ends. It can, for instance, be used if you want to skip a number of steps at the beginning of a simulation befor saving data because they are transients. Another reason is to set a global skip-interval on top of the temporal selections for the individually saved variables. That can be desirable if the amount of data generated is very big, but storing less steps would already be sufficient. To setup the timer use

```
SET_SAVE_TIMER( start, end, skip )

TIMER_ON
TIMER_OFF
```

SET_SAVE_TIMER only sets the parameters of the timer, i.e., the first and maximal step it tries to save anything, "start" and "end", and the interval (in simulation steps) at which data is stored, "skip". If it is to be used, the timer has to be enabled explicitly, either from the GUI by right-clicking on the Save-button and selecting the appropropriate tick-box or by calling TIMER_ON from the source code. It furthermore only generates output if the global save switch is on in addition (the master fuse for your valuable hard disk space).

Observe that the GUI also allows to set the parameters of the timer ("Stop Watch") by hand; they do not need to be set in the code.

5.1.6 Examples

```
int nx=3, ny=2;
int xsel[3]={1,2,5};
int ysel[2]={3,4};

BEGIN_OUTPUT

  SET_ITEM_SEPARATOR( "\n" )

  // 1. example

  OUTFILE("patterns")
    SAVE_VARIABLE( "pats", pats, ARRAY_INT_TYPE, Nones, P, ONINIT, 0, 0)
```

```
  // 2. example

  OUTFILE("Quality")
    SAVE_VARIABLE( "qual", &Q, FLOAT_TYPE, 0, 0, 0, 0, 0 )

  // 3. example

  OUTFILE("file42")
    SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON)
    SAVE_VARIABLE( "z", z, bVECTOR, N, 0, 0, 0, 0 )
    SAVE_VARIABLE( "phi1", pot1, MATRIX, xsize, ysize, 0,0,0 )

  // 4. example

  OUTFILE("phi2")
    SAVE_VARIABLE( "phi2", pot2, MATRIX, xsize, ysize,  SKIP | GRID ,
                   TSkip(2), Grid(38, xsize, 100, 32, ysize, 100) )

  // 5. example

  OUTFILE("phi3")
    SAVE_VARIABLE( "phi1", pot1, MATRIX, xsize, ysize, IRR_GRID
                   0, Irregular( nx, xsel, ny ysel ) )

END_OUTPUT
```

In the example the item separator is first set to \n such that individual entries go to separate lines.

The first example defines an output file "patterns" to which an integer array "pats" of size Nones × P is stored once after each call to the top-level init() function (selected by the ONINIT flag). There are no further spatial or temporal selections.

The second example stores a single floating point variable "Q" in each step to a file "Quality".

The third example – in contrast to all others – stores data in ASCII format because the respective flag is set. Data goes to a file "file42". Stored per step are a binary vector "z" of size N and a matrix "pot1" of size $xsize \times ysize$ without any further spatial or temporal restriction.

The fourth example stores a matrix "pot2" of size $xsize \times ysize$ to a file "phi2". Only every second time step is stored and the matrix is spatially sub-sampled on a regular grid.

The last example subsamples a matrix on an irregular grid, but there is no temporal selection.

5.2 Input

A graphical user interface for input from files is not available and not planned. Raw file input functionality has to be used instead (see next section).

5.3 Raw I/O

Instead of using the graphical interface for I/O operations, those can be included directly in the application program using the usual C file access options (see textbooks on C-programming).

Chapter 6

Felix Parameter Search & Sensitivity Module

The module psearch.c/h introduces some parameter search or scanning facilities into Felix. For a set of parameters regular grids or irregular sets of points can be defined. The module psearch then provides a multi-index that iterates through the Cartesian product. It is also possible that some search directions update several parameters at once. If for each parameter set several simulation runs are desired this can be specified, too.

(Spike train) metrics

Sensitivity

6.1 General Usage

It happens often that a simulation has to be executed many times with different parameters if a parameter space has to be scanned, or with the same parameter if data are collected for further statistical evaluation. The psearch-module supports this process. It allows to define various parameter dimensions together with range specifications for the values these parameters can take. It then implements a multi-index that iterates through all possible cartesian parameter combinations.

A typical usage scenario would be that the module is initialised in `main_init()` and that parameters that have to be scanned are not further changed in init(). If certain conditions are reached in the `step()`-function, e.g., after a fixed number of simulation steps, the next parameter set is selected. Although not necessary in general, the `init()`-routine can afterwards be called, if that is desired to re-initialise other parameters and variables. If the module has cycled through all parameter combinations the simulation can exit.

```
main_init()
{
  ...
  psearch_init();           // initialise internal structs
  psearch_add_param(...);   // add a param to scan
  ...                       // ... add more as required ...
}
```

```
...
step()
{
   ...  DO THE WORK ....

  if (SIM_TIME >= 100.)  // some criterion for finishing a single simulation
  {
    if ( psearch_next_param(); )  // get the next parameter set
      init();           // reset other parameters / variables
    else
      exit();           // psearch_next_param() returns 0 if we are done
  }
}
```

6.2 Parameter Scan Functions

6.2.1 Initialisation and setup

Before it is used the psearch module has to be initialised by calling `psearch_init()` in `main_init()`.

```
void psearch_init();
```

There can be only one set of scan-parameters per simulation.

After initialisation the set of parameters to scan/search is empty. Add parameters by using

```
void psearch_add_param( float*p, int type, int npoints, float*data );
 # define PSEARCH_RANGE     0x1
 # define PSEARCH_POINTS    0x2
```

Only floating point parameters are supported (integer-values can be emulated by floats). `p` is the address of the parameter to vary; we need the address such that we can change its value. `npoints` is the number of values the variable p is supposed to take in the scan. `type` can be either `PSEARCH_RANGE` or `PSEARCH_POINTS` which determines the meaning of the fourth data-argument:

- `PSEARCH_RANGE`: This type defines a regular grid of points. `data` must be a 2-dimensional array where `data[0]` is an offset and `data[1]` an increment. `p` takes values: `data[0]` + i*`data[1]`, for $i = 0, 1, 2 \ldots$ `npoints-1`.
- `PSEARCH_POINTS`: This defines an irregular collection of points. `data` contains `npoints` floating point values the parameter `p` will cycle through.

For further explanations see the example section below.

As parameters are added to the parameter set they are initialised to their lowest indexed value (which is not necessary their lowest value, if the increment in `PSEARCH_RANGE` is negative or the values in `PSEARCH_POINTS` are not ordered according to size).

6.2.2 Iteration through the parameter product space

To iterate through the cartesian product of the parameter sets in the individual parameter dimensions use the function

```
int  psearch_next_param();
```

This function takes account of whether a parameter has been defined as `PSEARCH_RANGE` or `PSEARCH_POINTS`. It sets the parameters internally to their new values. The function returns 1 if there was a parameter set left, and otherwise zero. Afterwards returning 0 the beviour of further calls to `psearch_next_param()` is undefined. Indeed, a return value of zero should in general trigger post-processing of data and exiting of the simulation.

6.2.3 Running multiple simulations for each parameter set

It can be desired to run a simulation several times for each parameter set. This can be reached by using

```
extern void psearch_set_repetitions( int k );
```

If `psearch_set_repetitions( int k )` sets `k` to a value bigger than 1 (the default) the parameters are only changed every k calls to `psearch_next_param()`.

6.2.4 Changing several parameters per search dimension

The mechanism so far apply to single parameters in each dimension. It is possible to define dimensions where more than one parameters are varied. This can be useful when the number of parameters is so high that a full search through the cartesian product space is unfeasible or if for some reason only values on a certain set of points in the full parameter space are needed, but not a complete cartesian sub-sample. The following function supports this functionality.

```
void psearch_add_nd_param( int n, float*p, int npoints, float*data );
```

Here, `n` is the number of parameters to modify and `p` a vector of parameters of length `n`. `npoints` is the number of sample points in the `n`-dimensional sub-space of parameters, and `data` is an array of `npoints` sample points of dimension `n`, i.e. `data[i*n+j]` is the value of parameter `p[j]` in the i-th sample point.

Note that higher-dimensional parameter spaces can be combined with the previously explained one-dimensional ones. This is demonstrated in the example in subsection ??.

6.2.5 Support functions to print indexes and parameters

There are a few support functions that print out information about the internal state of the module:

```
void psearch_print_data();
void psearch_print_params( FILE*file );
void psearch_print_index_string( char*string );
```

`psearch_print_data()` prints the internal data structures to the screen which is mainly useful for debigging purposes.

`psearch_print_params()` prints the actual parameter values to a stream. if `file` is "stdout" output goes to the screen, but it can also refer to a previously opened file, e.g., for book-keeping. The file must be open for writing, of course.

`psearch_print_index_string( char*string )` formats the internal multi-index that enumerates the cartesian product and prints it in ASCII-format to a string (which must be long enough and is not checked). If there are 3 parameters scanned the output could be `1-3-2-0`, meaning that the first parameter currently takes its first value from the range of possibilities, the second parameter its third value, and the third parameter its second value. The fourth number is the iteration for this particular parameter set (see `psearch_set_repetitions()` above). This function can be useful to construct filenames for data output.

6.3 Example: Scanning a parameter space

The example below (see "tst-psearch.c" in the code directory) shows how to set up a parameter scan with 3 dimensions, two single parameter dimensions where one parameter (`p1`) is sampled on a regular grid and the other one (`p2`) on an irregular set of points, and a third dimension consisting of a number of points (2) for two further parameters in `p[2]`.

```
# include <felix.h>

float p1, p2, p[2];
float data1[2]= { 2., .2 };
float data2[3]= {-1., 3., 7};
float data[4] = {1., 2., 3., 4.};
char str[100];

NO_DISPLAY
NO_OUTPUT

main_init()
{
  psearch_init();
  psearch_set_repetitions( 2 );
  psearch_add_nd_param( 2, p, 2, data );
  psearch_add_param( &p1, PSEARCH_RANGE,  2, data1 );
  psearch_add_param( &p2, PSEARCH_POINTS, 3, data2 );

  psearch_print_data();
}
```

```
init()
{
  printf("init() called\n");  // noting initialised here, but could be
}

step()
{

  if (SIM_STEP==1)  // print the current parameter set
  {                 // to screen, but only once
    psearch_print_index_string( str );
    printf("%s\n", str);
    psearch_print_params( stdout );
    printf("\n");
  }

  // do the hard work here
      //  ........


  if (SIM_STEP == 4) // after me steps ...
  {

    if (psearch_next_param()) // ... get the next parameter set
      init();   // re-init variables as desired
    else
      exit(0);  // or exit, if all parameter sets simulated

    SIM_STEP = 0; // need to reinit this;
                  // otherwise SIM_STEP == 4 stays false forever
  }
}
```

6.4 Interfacing parameter search and file output

Felix provides mechanisms to store simulation data to files, see section **??**. File output is activated on demand when the respective button in the GUI-version is pressed, and by default active in a non-GUI, i.e., parallel version. However, once opened, output goes to only one set of files as specified in the Felix-file, unless these files are reset. This is possible by hand in the GUI. For an automatic parameter search, however, one would usually prefer a non-interactive file-reset (unless one wants to have all output directed to the same files even for different parameter sets).

One solution would be to code the file output explicitly into the Felix-application, i.e., open files per new parameter set and save data explicitly.

More convenient is the use of the "template-feature" of the file name generation routines (NOTE: this is a feature currently undocumented in the I/O-section). By default when files are opened, Felix takes the basename as defined in the `OUTFILE("basename")` specification. The template-

feature allows to append the base-names by templates, e.g., for different parameter sets.

The template can be set by `SetSaveTemplate( str )` where `str` is the tmplate string.

To make a template active, the actually open output files have to be closed and reinitialised. This is done by calling `InitOutFiles();`.

The function `psearch_print_index_string( str )` described in section **??** sets a string to a representation of the the current multi-index of an iteration through a parameter set. It can be conveniently used as a template.

Below is an example, where the noise in an integrate and fire neural network is varied. Unnecessary code has been cut away. The complete file can be found in the documentation code directory (inf2scan.c).

```
...

float psigma;                // parameter for the scan
float offsinc[2]={0.,.05};  // offset and increment (for PSEARCH_RANGE)
char str[16];

...

SliderValue ssigma = 0;    // slider ssigma UNUSED here !!


BEGIN_DISPLAY
  ...
END_DISPLAY


BEGIN_OUTPUT
  SET_ITEM_SEPARATOR( "\n" );  // newlines after spikes
  OUTFILE("spikes")
    SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON ) // readable format
    SET_SAVE_FILE_FLAG( THISFILE, GDF, ON )  // only spike times stored
    SAVE_VARIABLE( "out", z,  bVECTOR, N, 0, 0, 0, 0 )
END_OUTPUT


int main_init()
{
  ....

  psearch_init();                    <<<<< initialise the parameter scan
  psearch_set_repetitions( 2 );      <<<<< 2 repetitions per parameter set
  psearch_add_param( &psigma, PSEARCH_RANGE,  4, offsinc );  << 1 parameter
}

int init()
```

```
{
  .... // don't change psigma in init() !!! (unless you know what you do)

  psearch_print_index_string( str );   <<<<<<< setup template string
  SetSaveTemplate( str );              <<<<<<< and store as template
  InitOutFiles();                      <<<<<<< reinit the file ("spikes")
}


int step()
{
  int i;

  for (i=0;i<N;i++)  // leaky integration for all neurons
    leaky_integrate ( tau, x[i],
        0.01*( sI + sJ0*v[i]) + psigma*gauss_noise() );  <<<<<< psigma!
  Fire_Reset( N, x, 1.0, 0.0, z );  // firing and reset
  bMult( N, N, J, z, v ); // redistribution spikes

  if (SIM_STEP == 1000)          <<<<< terminate current simulation run
  {
    if (psearch_next_param())    <<<<< work left ? next parameter set
    {
      SIM_STEP = -1;             <<<<< need to reset
      init();                    <<<<< reinit; including template & files
    }
    else
      exit(0);                   <<<<< finished ...
  }
}
```

NOTE: The location of the reinitalisation of SIM_STEP can be crucial. Done in the init() routine it can lead to a one-step offset of the first simulation run compared to the others.

6.5 Parameter Sensitivity of Simulations

"Sensitivity analysis" can provide insight into the parameter dependences of a simulation, ie., whether the simulated dynamical patterns vary much if some parameters are changed, which parameters or parameter combinations have the strongest impact, and which are not so important at all, because activation patterns hardly depend on them.

This type of analysis in general needs some measure to compare different simulation runs. A general class of such measures with well defined mathematical properties are so-called "metrics". Those consider simulated trajectories (e.g., potential traces, single or multiple unit spike-trains) as points in an abstract space, a so-called metric space, and define how to compute distances between these points. Changing one (or a set) of parameters will change the simulated activity and thereby the location of the point representing it in the metric space. Parameter combinations that change

the location a lot are sensitive, those that have hardly any impact insensitive.

Note that this provides a local characterisation of sensitivity only, as changes are relative to some fixed set of parameters. A global analysis is usually much more difficult to do and regularly requires exhaustive exploration of the parameter space (with the exception of a few simple or fortunate cases).

Furthermore, the sensitivity properties of a simulation can very much depend on the precise metric chosen. For example, metrics exit that value the precise location of single spikes, whereas others only operate on instantaneous firing rates. For more information see the next section.

6.5.1 Spike-train and other metrics

... to come .....

6.5.2 Sensitivity Measures

Given a simulation program with observables x and parameters p, a default set of parameters p^*, and a metric $d(x, y)$ on the observables, the sensitivity of the model with respect to the default parameters and metric chosen can be studied.

One way to determine parameter sensitivities is to compute the gradient (if it exists) of the distance from the default point with respect to the parameters: $\nabla_p d(x(p), x(p^*))$. The gradient of a function of some parameters is a vector in parameter space, that points into the direction of the parameter combination that changes the value of the function most. In our case the function is the difference between the activation pattern given the default parameters and those for any other set of parameters close by. Large (absolute) entries in the gradient indicate a strong impact on the simulated patterns by the respective parameter. However, note that the scale (or units or measurement) of the parameters can vary, and that not all parameters can be easily compared with each at all. Sometimes parameter changes are therefore "normalised" by their absolute value before comparison, but this can also fail, if the default value for some parameters is zero or close to it.

6.5.3 Gradient Computation

The parameter search module can be conveniently used to generate simulations for a subsequent gradient analysis. This requires computing activation patters for the default paremeter set as well as for small changes in the various parameter directions. If the parameter changes are "small" the perturbed patterns can be used to compute approximations of the partial derivatives in the direction of the respective parameter. Let's say there are m parameters. The partial derivatives are then the components of the full gradient: $\partial d(x(p) - x(p^*))/\partial p_i = \partial d(x(p_1^*, \ldots, p_i^* + \Delta_i, \ldots, p_m^*) - x(p^*))/\partial p_i \approx d(x(p_1^*, \ldots, p_i^* + \delta_i, \ldots, p_m^*) - x(p^*))/\Delta_i,\ i = 1 \ldots m$.

The following Felix function prepares $m + 1$ parameter vectors for the computation of activity patterns for the default parameter set $x(p^*)$ and small perturbations in the m parameter dimensions, $x(p_1^*, \ldots, p_i^* + \delta_i, \ldots, p_m^*$. It returns an $(m + 1) \times m$ matrix of parameter sets for use with `psearch_add_nd_param()`, the parameter space scan function that operates on higher-dimensional

sets of points but not their full Cartesian product. The first parameter set in the matrix is for the default parameter set, the remaining ones for directions $i = 1 \ldots m$.

```
float*setup_grad_params( int m, float*params, float eps, float*delta )
```

m is the number of parameters and *params* is the vector of parameters. *eps* is a small number and *delta* is an m-dimensional vector of perturbations. If *delta* is non-zero, the perturbation in direction i is $eps\delta_i$. If it is zero, all perturbations Δ_i are equal to *eps*. If all the parameters obtain values on the same scale the use of just a single value *eps* for the perturbations is more convenient than defining a full vector of perturbation Δ.

After preparing the set of parameter settings in `main_init()` the parameter search module has to be initialised and run precisely in the way described in section ?? for n-dimensional parameter sets, `psearch_add_nd_param()`. The program will then iterate through the simulations for the default parameter set and the m perturbed parameter sets.

Data of the simulations can be output to files for subsequent computation of distances and gradients. As mentioned initially the sensitivity may depend crucially on the particular metric chosen. It it therefore often preferable not to compute in the simulation program, unless the best metric is know in advance.

6.5.4 Example: Gradient computation

There is a program inf2grad.c in the documentation's code directory that shows how to run simulations for a gradient computation.

```
# include <felix.h>

# define N   100    /* number of neurons      */
# define tau  10.   /* membrane time constant */

Vector  x;          /* potentials             */
Matrix  J;          /* connections            */
bVector z;          /* vector of spikes       */

Vector  v;          /* auxiliary variable     */

int    nparams = 3;
float  pars[3] = {101., 10., 0.}, // pI, psigma, pJ0;
       delta[3]= { 10.,  1., 0.1};
float *paramsets;
float noiseseed;

BEGIN_DISPLAY
...
END_DISPLAY
```

```
BEGIN_OUTPUT

  SET_ITEM_SEPARATOR( "\n" );

  OUTFILE("spikes")
    SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON )
    SET_SAVE_FILE_FLAG( THISFILE, GDF, ON )
    SAVE_VARIABLE( "out", z,  bVECTOR, N, 0, 0, 0, 0 )

END_OUTPUT


int main_init()
{
  noiseseed = time(NULL);            //  <<<<<<<<<
  randomize( noiseseed + 123456 );   //  <<<<<<<<<
  SET_STEPSIZE( .1 )

  J = Get_Matrix( N, N );
  x = Get_Vector( N );
  z = Get_bVector( N );
  v = Get_Vector( N );

  paramsets = setup_grad_params( nparams, pars, .1, delta );   // <<<<<<<<

  psearch_init();                                              // <<<<<<<<
  psearch_add_nd_param( nparams, pars, nparams+1, paramsets ); // <<<<<<<<
}

int init()
{
  int i;
  char str[16];

 // randomize( noiseseed );           //  <<<<<<<<<<

  Clear_bVector(N,z);
  Clear_Vector(N,v);

  for (i=0; i<N; i++)
    x[i] = equal_noise();
  Make_Matrix( N, N, J, 1.0/N, .4/N  );

  psearch_print_params( stdout );

  psearch_print_index_string( str );
  SetSaveTemplate( str );
  InitOutFiles();
}
```

```
int step()
{
  int i;

  for (i=0;i<N;i++)  // leaky integration for all neurons
    leaky_integrate ( tau, x[i],
            0.01*( pars[0] + pars[1]*v[i]) + pars[2]*gauss_noise() );
  Fire_Reset( N, x, 1.0, 0.0, z );  // firing and reset
  bMult( N, N, J, z, v ); // redistribution spikes

  if (SIM_STEP == 1000)
  {
    if (psearch_next_param())
    {
      SIM_STEP = -1;               // <<<<<<<<<<
      init();
    }
    else
      exit(0);
  }
}
```

An important note regarding simulations with noise are at hand, cf., the lines in the code above indicated by <<<<<<. Apparently, even if parameters are identical, simulations with noise can potentially lead to very different activation patterns. It can therefore be necessary to reinitialise the random number generator each time the simulation is restarted. Some lines indicated in the `main_init()` and `init()` routines above show how to do this. However, even with a reset of the random number generator comparability is not necessarily guaranteed as the executed code can contain subtle interactions between parameters and the noise generation. In such cases it can be required to precompute random sequences and reuse them in the iterations through the parameter space.

Chapter 7

The Felix MIDI Interface

This document describes the use of the new Felix-MIDI-interface. It is merely a collection of notes as the interface is "in progress". A number of examples are discussed. Changes in the future are likely.

7.1 Introduction

The strategy used to implement MIDI functionality in Felix is the following: A MIDI interface should have at least an output, i.e., readable MIDI port, which sends events to some sound-generator, but preferable also an input interface, i.e. a writable port, which may connect it to a keyboard. Both options are provided in the preliminary implementation. If a simulation runs which initialises one or both ports, they can be connected to other devices by means of readily available Linux software. If the computer used has access to a hardware MIDI device (e.g., in the soundcard or connected to the usb port) this can be accesses, too.

In a running Felix simulation NOTE_ON and NOTE_OFF MIDI-events can be issued asynchronously, for instance, by neurons that spike. The events are directly scheduled; a MIDI-queue is not used at the moment. The user has to provide code that emits the events to the readable port. MIDI-channel, key, and velocity can be specified.

On the input side an event-receiver can be (optionally) started, which is spawned in a separate thread in order not to block simulations; it waits for events on the writable port. The user has to translate the incoming events into inputs for his/her Felix application.

For testing I use a setup with a virtual keyboard (vkeybd), a software synthesiser ("qsynth" – a graphical frontend to fluidsynth), and the (software) switchboard "qjackctl" which is a frontend to "jack", a Linux audio environment. qjackctl is used to connect the ports of the devices/programs in the MIDI-environment. Note that qjackctl and qsynth must be run as root:

1. start the Jack switchboard: "sudo qjackctl"
2. press the "start" button in qjackctl - this should start the jack-demon which serves your MIDI requests
3. start the virtual keyboard: "vkeybd" (not necessarily as root)

4. start the synthesiser: “sudo qsynth”

5. start your Felix program

6. in qjackctl press “connect” and connect the keyboard to the writable Felix port (if present) and the Felix readable port to the synthesiser, see Figure 7.1.

7. Press the run-button in Felix to start the simulation (required to schedule the note-events)

8. For a test without Felix you can also connect the keyboard directly to the synthesiser, in which case pressing a key should result in an audible tone. If it does not, your software MIDI environment is not properly setup.

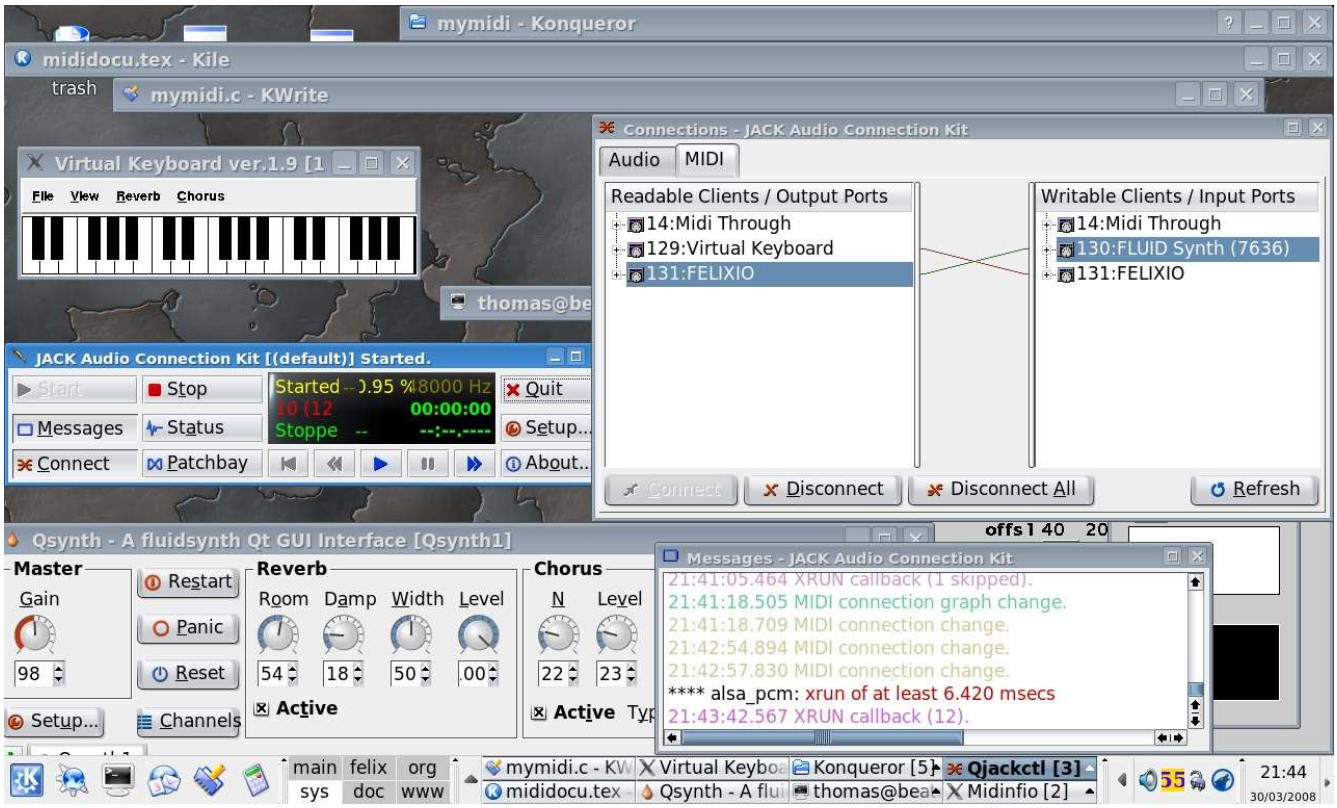

Figure 7.1: Felix-MIDI setup under Kubuntu. The Felix program is hidden under the visible windows.

7.2 Functions provided by mymidi.o

7.2.1 Compilation

Some of the programs in this documentation don’t need the full Felix package, but just the mymidi.o library. If the name of such a program is `<expl>.c` it is compiled with `gcc -o <expl> <expl>.c mymidi.o -lasound -lpthread`. Start the program on the command line and connect it using “qjackctl”.

A Felix program with MIDI interface is compiled in the usual way: `Felix <expl>`. Run and connect the program as explained above.

7.2.2 Initialisation

The following functions initialise an interface to the sequencer and open readable and writable ports:

```
snd_seq_t *open_seq( snd_seq_t *seq_handle, char*basename );
int create_readable_port( snd_seq_t *seq_handle,
                                  char*basename, char*ext);
int create_writable_port( snd_seq_t*seq_handle,
                                  char*basename, char*ext);
```

- `seq_handle` is a handle to the sequencer interface
- `basename` is the name under which the Felix application appears in the MIDI environment
- `ext` are extensions to the basename that might be useful to distinguish different ports

7.2.3 Setting up an event loop

If an application has an input port (a writable port) it can receive MIDI-events. This is typically done in a loop that waits for the events and calls a user supplied routine for each incoming event. The following two routines implement this functionality:

```
int enter_event_loop(snd_seq_t *seq_handle,
           int midi_action(snd_seq_t *, snd_seq_event_t *ev) );
int midi_action_print_event(snd_seq_t *seq_handle, snd_seq_event_t *ev);
```

`enter_event_loop` implements the main loop; it requires a sequencer handle as an argument and a second function that defines what to do with the events.

`midi_action_print_events` is an example for an event-handling function. It prints information about an event to the screen together with some information about parameters (channels, keys, etc.). It does not handle all possible event types (see appendix 7.5.1). You can use it as a prototype for your own event-handlers.

Note that the function `enter_event_loop` iterates an infinite loop until the midi_action()-function returns a negative value on some event. This means you can't use it directly in a Felix program, because you also need to step through the simulation. A solution for this problem is to run the MIDI-receiver in a separate thread as will be explained later. The next sub-section presents a non-threaded (and non-Felix, just C) example.

7.2.4 A first example

The C-code below opens a virtual sequencer device with in- and out-ports. It then enters a loop that waits for incoming events and prints them to the screen.

```
# include <stdio.h>
# include <stdlib.h>

# include "mymidi.h"

main()
{
  snd_seq_t *seq_handle;
  char*basename="MIDITST";

  // open ALSA sequencer device
  seq_handle = open_seq( seq_handle, basename );

  // setup output port
  create_readable_port(seq_handle, basename, "rd-1");

  // setup input port
  create_writable_port(seq_handle, basename, "wr-1");

  // Setup ALSA event loop
  enter_event_loop( seq_handle, midi_action_print_event );
}
```

Compile, run, and connect this program to the virtual keyboard (vkeybd) as explained earlier. It should print the pressed notes to the screen (onsets and offsets). Note that the program opens an output port, too, which however, is not further used at all.

7.2.5 Sending note events

In a running program, events can be most easily scheduled asynchronously meaning that they are directly forwarded to the sequencer without getting queued. The following two functions send onsets and offsets of notes to some sequencer interface and port. The channel, key and velocity values can also be specified.

```
send_noteon(snd_seq_t *seq_handle, int port, int ch, int key, int vel);
send_noteoff(snd_seq_t *seq_handle, int port, int ch, int key, int vel);
```

There are two further functions mainly for debugging purposes in the event handler:

```
show_note(snd_seq_ev_note_t*note);
show_sequencer_event(snd_seq_event_t *ev);
```

The first of these functions prints details about a Note, the second about a whole event.

7.2.6 Threaded event receivers

In sub-section 7.2.3 we set up a simple event receiver, but mentioned that in a Felix program we need not only receive events but also drive the simulation continuously. A straightforward way to satisfy both requirements at the same time is to split the program into two parts and execute them in two so-called "threads". These are light-weight processes that can do work independently. The following function allows to spawn a thread from the main program that enters a MIDI event loop and executes the function `midi_action` per received event.

```
int start_midi_receiver( snd_seq_t * seq_handle,
                 int midi_action(snd_seq_t *, snd_seq_event_t *ev)  );
```

The function listens on all writable ports that are attached to the sequencer device `seq_handle` . The function `midi_action` has the same prototype and behaviour as for a non-threaded event loop, see subsection 7.2.3 and the example in subsection 7.2.4. It is therefore possible to use the same event handler functions, for instance the simple event printout function `midi_action_print_event` used in example 1 (see appendix 7.5.1 for the full code of this function). Example 2 in the next sub-section follows this approach. All events that are not handled in the `midi_action` function are discarded.

7.2.7 Example 2: A threaded MIDI receiver

This program opens a sequencer device with a writable (input) port. It then spawns a thread that listens for incoming MIDI events and deals with them using the same function as used in the event handler of the non-threaded example `midi_action_print_event` in sub-section 7.2.4; it does nothing but printing out some information about the incoming events.

```
# include <stdio.h>
# include <stdlib.h>
# include <pthread.h>

# include "mymidi.h"

# define BASENAME "RECEIVE-EVENTS"

int main (int argc, char *argv[])
{
  snd_seq_t *seq_handle;

  seq_handle = open_seq ( seq_handle, BASENAME );
  create_writable_port( seq_handle, BASENAME, "" );

  start_midi_receiver( seq_handle, midi_action_print_event );

  while (1)
  {
    usleep( 1000 );
```

```
  }

  exit(0);
}
```

Observe that after having spawned the MIDI-receiver-thread the main program enters an infinite loop. In the example it just sleeps for a short time in each iteration. In a more useful application the loop would contain some code to compute, e.g., a neural simulation.

7.2.8 Simple MIDI startup

Many applications can probably just live with a single input and a single output port. Initialisation of such a setup can be done using the following function and is demonstrated in the Felix example in section 7.3

```
snd_seq_t * init_simple_midi( char*basename, int *port,
                  int midi_action(snd_seq_t *, snd_seq_event_t *) )
```

The function expects a "basename" under which it appears in the MIDI environment and an event handler function like `midi_action_print_event`. It returns the handle to the sequencer and the port number for sending events to other devices.

7.3 A Felix application

Example 2 in subsection 7.2.7 demonstrates the basic way how MIDI is integrated into Felix: A receiver-thread has to be spawned in the maininit() function of a Felix-program that deals with incoming events if that is desired. The main program can then continue with the Felix simulation in the usual way, i.e., the Felix-step-routine can then do the main work of the simulation. The step-routine will typically also schedule *output*-events send to a readable port, say, connected to a synthesiser or other device. Because the program generates the output events itself it does not need a thread to wait for them as well. They can just be issued as required.

Here is an example that implements an integrate and fire neuron network (derived from the default Felix example "inf.c") with in- and output. It reads events from an input port, say, connected to the virtual keyboard *vkeybd*. Because vkeybd has 36 keys, 36 integrate and firing neurons are used. ON and OFF events for certain keys determine whether the respective neuron receives and extra input or not. The program also write output to a port. If a neuron spikes it sends a NOTEON-event followed by a NOTEOFF and a key-number corresponding to its index plus some offset. The lower half of the neurons send to a channel specified by slider "sch1" and the upper half to "sch2". The offsets can be independently changed using the sliders "soffs1" and "soffs2" - one instrument can this way play very low-pitch notes and the other one high notes. The velocity can further be changed using slider "svel".

```
// midinfio.c  - integrate and fire neural network
//               with MIDI output and input
```

```
# include <felix.h>
# include <mymidi.h>

# define BASENAME "FELIXIO"
snd_seq_t *seq_handle;
int port;

# define N 36    // number of neurons = # keys in vkeybd

Vector  pot,v1, midin;
Matrix  J;
bVector o, o1;

SliderValue snoise =  10;
SliderValue sinput = 105;
SliderValue smidin = 105;
SliderValue sJ     =  50;
SliderValue sch1   =   0;
SliderValue sch2   =   9;
SliderValue svel   = 100;
SliderValue soffs1 =  40;
SliderValue soffs2 =  40;

BEGIN_DISPLAY

 SLIDER( "noise",    snoise, 0, 100)
 SLIDER( "input",    sinput, 0, 200)
 SLIDER( "midi in",  smidin, 0, 200)
 SLIDER( "coupling", sJ,     0, 200)
 SLIDER( "ch 1",     sch1,   0,  15)
 SLIDER( "ch 2",     sch2,   0,  15)
 SLIDER( "velo",     svel,   0, 255)
 SLIDER( "offs1",  soffs1,  20, 120)
 SLIDER( "offs2",  soffs2,  20, 120)

 TIMER(100)

 WINDOW("time courses")
  IMAGE(  "pot", AR, AC, pot, VECTOR,  6, 6, 0.0, 1.0, 15)
  RASTER( "pot", NR, AC, pot, VECTOR,  N, 0, 0.0, 1.0, 1)
  GRAPH(  "pot", NR, AC, pot, VECTOR,  N, 0, 0, 0, -.01, 1.01 )
  RASTER( "out", NR, AC,   o, bVECTOR, N, 0, -.01, 1.01, 2)

END_DISPLAY

NO_OUTPUT

// define what to do with incoming events
```

```
int midi_action(snd_seq_t *seq_handle, snd_seq_event_t *ev)
{
  int cc = 48; // index of lowest key in vkeybd
  switch (ev->type)
  {
    case SND_SEQ_EVENT_NOTEON:
      if ( (ev->data.note.note - cc >= 0)
            && (ev->data.note.note - cc < N)  )
        midin[ev->data.note.note - cc ] = 1.;
      break;

    case SND_SEQ_EVENT_NOTEOFF:
      if ( (ev->data.note.note - cc >= 0)
            && (ev->data.note.note - cc < N)  )
        midin[ev->data.note.note - cc ] = 0.;
      break;
  }
  return 0;
}

int main_init()
{
  randomize( time(NULL) );
  SET_STEPSIZE( .05 )
  J     = Get_Matrix( N, N );
  pot   = Get_Vector( N );
  o     = Get_bVector( N );
  o1    = Get_bVector( N );
  v1    = Get_Vector( N );

  midin = Get_Vector( N );
  seq_handle = init_simple_midi( BASENAME, &port, midi_action );

  return 0;
}

int init()
{
  int i;
  Clear_Vector(N,midin);
  Clear_bVector(N,o);
  Clear_bVector(N,o1);
  Clear_Vector(N,v1);
  for (i=0; i<N; i++) pot[i] = equal_noise();
  Make_Matrix( N, N, J, 1./N, 4./N );
  return 0;
}

int step()
```

```
{
  int i;

  for (i=0;i<N;i++)
    leaky_integrate( 1., pot[i],
             0.01*( sinput + sJ*v1[i] + smidin*midin[i]
               + snoise*gauss_noise() ) );
  Fire_Reset( N,  pot, 1.0, 0.0, o );
  bMult( N, N, J, o, v1 );

  // send left half spikes to channel 0; right half to 1
  for(i=0; i<N; i++)
  {
    int ch, note;

    if ( i < N/2 ) {
      ch = sch1;
      note = soffs1 + i;
    } else {
      ch = sch2;
      note = soffs2 + i - N/2 ; }

    if (o[i] > o1[i])      // note on
      send_noteon( seq_handle, port, ch, note , svel );
    else if (o[i] > o1[i]) // note off
      send_noteoff( seq_handle, port, ch, note, 0 );
    o1[i]=o[i]; // save value for next step (on/off detection)
  }

  return 0;
}
```

7.4 Sending Events over a local network

7.4.1 Local Network Routing – dmidid

Although it is planned to extend the Linux ALSA sound packages to connect devices not only on the local machine but also over a network, this functionality is not yet implemented.

However, the WWW provides some links to LAN-enabled MIDI. I have experimented with "dmidid" a protocol and C-code that uses raw sockets (see http://www.dimid.org; warning: 95% advertisements). Raw sockets are quite fast because they are implemented just on top of the physical network device layer. Thereby they bypass some potentially time-consuming TCP/IP processing. The latter includes any firewall, which might or might not cause security issues. dmidid is further given prioritised scheduling for faster execution (it therefore needs root rights to run: "sudo dmidid <params>").

That said, the dmidid demon implements a kind of MIDI-over-ethernet router. Several demons can be run on the various computers in a local network where they each open one readable and one writable MIDI port for connections. They are visible in qjackctl and an be connected there as any usual local port. However, the demons in addition listen on the internet interfaces of the computers for incoming messages, and they can send messages themselves. Messages received for a MIDI-port on the local machine are routed to the respective device.

Messages are just written to the ethernet interface and received by any other interface on the local domain. The MIDI sender and receiver for the communication are therefore included in the transmitted internet packages, such that applications can filter packets addressed to them. Broadcasting is also possible.

The dmidid.c program in the Felix/mymidi distribution is modified from the original dmidid.c because the latter didn't allow to connect via the network to the local machine itself. That was needed for testing. An additional command line argument for the ethernet device to use has also been added.

Syntax:

```
dmidid [-v] [-b] [-i iface] [-t xx:xx:xx:xx] [-r xx:xx:xx:xx]
```

- -v prints the version
- -b sets broadcast mode (receive messages to ff:ff:ff:ff and myself)
- -i sets the interface to use (default "eth0")
- -t xx:xx:xx:xx is where I send to (default ff:ff:ff:ff)
- -r xx:xx:xx:xx is my receiver id (default ff:ff:ff:ff)

E.g. `dmidid -i eth1 -r 90:00:00:00 -t 90:00:00:00` starts a demon listening on eth1 (on laptops often the wireless device) with receiver id 90:00:00:00 and the same transmitter id, i.e., it sends to itself. If the broadcast flag is set when the demon is started, it also receives broadcast events (to ff:ff:ff:ff by another demon).

For more info see the original dmidid-package (http://www.dmidid.org).

7.4.2 MIDI over LAN

Communication between dmidid-demons is restricted to the local domain of the ethernet interface they are bound to; the packages are not routed to other networks. An interface that uses TCP/IP, i.e., the transport level, is under debelopment. It might have rather long response times and may therefore not be well suited for real-time applications, especially when they are closed-loop.

"Ping"-round-trip times to the COLAMN computer cluster are quite short (<5ms). There is an additional step from the master to the nodes. Because of the UoP firewall settings we may even be forced to use tunnels....

7.5 Appendices

7.5.1 Appendix 1 – The midi_action_print_event function

The following code shows the library function `midi_action_print_event` which prints events to the screen but can be used as a prototype for more interesting event-handlers.

```
int midi_action_print_event(snd_seq_t *seq_handle, snd_seq_event_t *ev)
{
  show_sequencer_event(ev);

  switch (ev->type) {
    case SND_SEQ_EVENT_CONTROLLER:
      fprintf(stderr, "Control event on Channel %2d: %5d       \n",
              ev->data.control.channel, ev->data.control.value);
      break;
    case SND_SEQ_EVENT_PITCHBEND:
      fprintf(stderr, "Pitchbender event on Channel %2d: %5d   \n",
              ev->data.control.channel, ev->data.control.value);
      break;
    case SND_SEQ_EVENT_NOTEON:
      fprintf(stderr, "Note On event on Channel %2d: %5d       \n",
              ev->data.control.channel, ev->data.note.note);
      break;
    case SND_SEQ_EVENT_NOTEOFF:
      fprintf(stderr, "Note Off event on Channel %2d: %5d      \n",
              ev->data.control.channel, ev->data.note.note);
      break;
  }
  return 0;
}
```

7.5.2 Appendix 2 – snd_seq_event_t and snd_seq_ev_note_t

```
/** Sequencer event */
typedef struct snd_seq_event {
snd_seq_event_type_t type; /**< event type */
unsigned char flags; /**< event flags */
unsigned char tag; /**< tag */

unsigned char queue; /**< schedule queue */
snd_seq_timestamp_t time; /**< schedule time */

snd_seq_addr_t source; /**< source address */
snd_seq_addr_t dest; /**< destination address */

union {
snd_seq_ev_note_t note; /**< note information */
```

```
snd_seq_ev_ctrl_t control; /**< MIDI control information */
snd_seq_ev_raw8_t raw8; /**< raw8 data */
snd_seq_ev_raw32_t raw32; /**< raw32 data */
snd_seq_ev_ext_t ext; /**< external data */
snd_seq_ev_queue_control_t queue; /**< queue control */
snd_seq_timestamp_t time; /**< timestamp */
snd_seq_addr_t addr; /**< address */
snd_seq_connect_t connect; /**< connect information */
snd_seq_result_t result; /**< operation result code */
snd_seq_ev_instr_begin_t instr_begin; /**< instrument */
snd_seq_ev_sample_control_t sample; /**< sample control */
} data; /**< event data... */
} snd_seq_event_t;


/** Note event */
typedef struct snd_seq_ev_note {
unsigned char channel; /**< channel number */
unsigned char note; /**< note */
unsigned char velocity; /**< velocity */
unsigned char off_velocity; /**< note-off velocity;
                                        // only for #SND_SEQ_EVENT_NOTE */
unsigned int duration; /**< duration until note-off;
                                        // only for #SND_SEQ_EVENT_NOTE */
} snd_seq_ev_note_t;
```

Chapter 8

Felix Remote Control and Data Streaming over Internet

Preliminary attempts have been made to give Felix an internet interface. At the moment it is possible to connect a running simulation to a telnet client providing a shell-like interface that allows to issue simple control commands like changing the speed of the simulation (via a timer), reinitialising it, or printing and changing the switch and slider values. It is also possible to open sockets and stream data to the internet that could be received by another application on a different computer.

It is planned, but not yet possible, to use Felix programs as remote controlers of Felix simulations, ie., to have a Felix-style interface that automatically sends slider and switch changes and receives output data which it displayed immediately.

Furthermore, at the moment it is only possible to remote control Felix programs without a graphical user interface. There are several reasons for this: 1) I mainly want to use the functionality for programs on a computer cluster, where programs have nio GUIs. 2) receiving asynchonous messages on sockets can hangup X11 badly; I am still trying to make the respective code stable; 3) For remotely controlling a simulation on another laptop or desktop it seems easier to just use desktop sharing (Krfb) or remote desktop connection facilities like Krdc or VNC.

8.0.3 Simulation Client Functionality

It seems natural at first to give a simulation *server*-functionality such that a remote control program can log into it in order to observe its activity and potentially modify its parameters.

However, on computer clusters jobs are often distributed to the compute nodes by special purpose software, so-called job queing systems. These schedule jobs as soon as appropriate resources become available. A problem on computer clusters with such scheduling queues is, that you won't know beforehand on which nodes your processes will run (Fig. 8.1 depicts a common situation). Therefore, it would be uncomfortable to log into a simulation by hand or automatically - you would first have to find out the node to connect to. There is furthermore not much control over the exact startup time of a simulation, meaning that it is hard to find out the node to connect to, connect, and then not miss the first so-and-so many thousand steps, when the simulation starts running immediately, which it should do because otherwise the fine computer cluster resources are wasted

by your program doing nothing but blocking the respective compute nodes it got allocated.

Therefore Felix simulations serves as a clients and not servers; they by itself connect to a remote server when they start up. The remote program can be listening on a fixed machine and a well specified port.

8.0.4 Meeting points

Two other problems on computer clusters concern their internet connectivity, see Fig/ 8.1.

1) The compute nodes are hardly ever visible from the outside world; only the master node is. Quite commonly internet packets from the compute nodes are not even routed towards locations outside the local network on the cluster.

2) It can furthermore be that your cluster is behind a firewall over which you have little or no control. This means only a restricted number of ports will be available for connections to the master node. However, typically at least the secure shell port (ssh, 22) will be open, because users need it to log into the cluster, and this port can be considered being safe, because ssh is well tested and implements high security standards.

These two problems - hidden compute nodes and firewalls blocking internet ports - complicate connecting Felix simulations on computer clusters to remote programs.

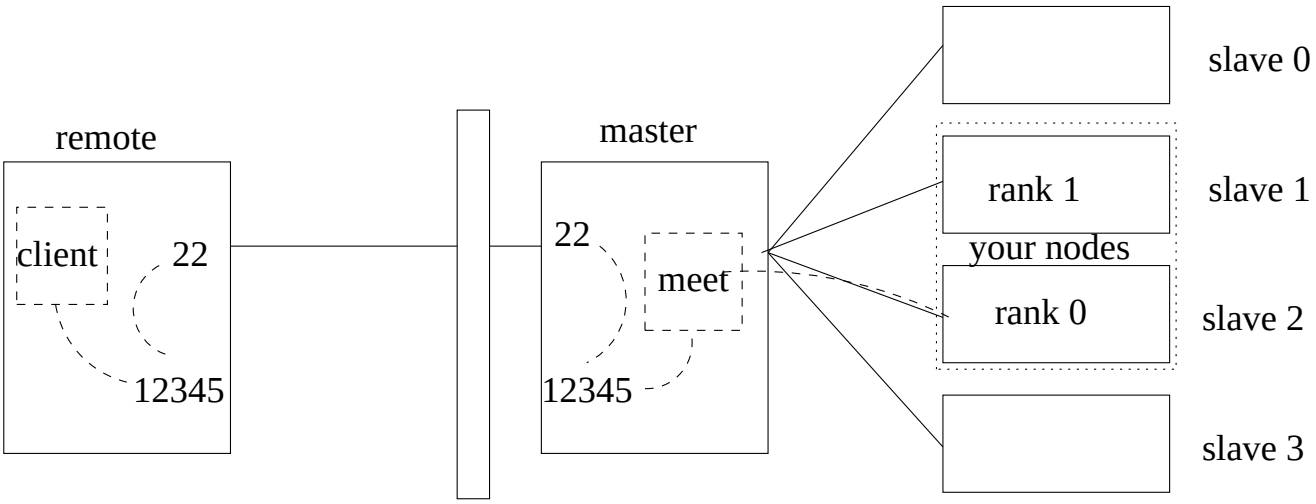

Figure 8.1: Typical remote control situation through a firewall. The job queue on the master schedules your job to random slave nodes. Rank 0 is assumed to connect to the meeting point which is listening on the master at port 12345 (on the internal and external interfaces!)). The client on the remote machine further makes a connection to local port 12345 and from there through a tunnel via ssh ports 22 adn the internet to port 12345 on the master. The tunnel can pass the firewall because ssh is necessary for the users to log into the cluster. This way a bidirectional communication line is setup.

As a solution to problem 1) we have implemented a simple "internet software router" that provides a "meeting point" where two processes can connect to and any traffic is bidirectionally routed. Such a meeting point would run on the master node of the computer cluster. A Felix simulation

that starts on a compute node can connect to the router because the master is on the local network of the compute node and has LAN-connectivity to the client. If there is no firewall (bad idea!) or the firewall has a hole punched at the port the meeting point is listenting on (also bad!), then a remote program can connect to the meeting point directly and communicate with the connected simulation. (In fact, the implemented meeting point software also allows reconnections. If only one side of the meeting point is connected everything that is sent to it will be silently dumped. The same or different clients can connect an arbitrary number of times during the runtime of a simulation.)

The above strategy is not very safe, because everybody can connect to your meeting point if it is publicly visible to the internet; there is no password protection; and the implementation of the meeting point might not even be secure, potentially giving crackers ways to break into your master node.

It is therefore better to connect to the meeting point via a secure internet tunnel. This is also a solution for restrictive firewalls, the second problem mentioned above and with respect to connections from the internet it is as safe as the ssh-protocoll is. Lucky enough it is not difficult to setup a tunnel; all that is needed is an open secure shell port (22, and of course a running sshd server connected to it).

However, just to mention it, from the security point of view, using a tunnel to connect to a meeting point on the master still leaves the possibility, that some other users of the cluster connects to your simulation. There is currently no way to prohibit this, but it might be that a future version of the meeting point program will have some password protection.

The meeting point program (meet) should be in the "tools"-directory of the Felix package.

The remote client to connect to the meeting point can be just the standard program "telnet" (because we connect through a tunnel the use of telnet is safe).

To connect a Felix simulation and a remote program follow these steps:

1. Start the meeting point on the master node: `meet <port>` where `<port>` is the port number to listen on. In the sequel we assume it is port 12345.

2. Compile your Felix program such that it connects to the meeting point when started. Only one connection should be made, ie. by rank 0 in an MPI application. You tell the Felix-kernel you want a remote connection by adding a statement `REMOTE( host, port )` somewhere in main_init().

3. Setup a secure shell tunnel from the remote computer to the port 12345 on the master node. If `<account>` is your acount name on the master, and xxx.xxx.xxx.xxx is the public IP address of the master, call on the remote machine:
   `ssh -N -L 12345:localhost:12345 <account>@xxx.xxx.xxx.xxx`

4. Start the client on the remote machine; e.g. `telnet localhost 12345`

5. Start the job on the master

8.1 Remote Connection Functionality

A remote connection to a Felix program provides a simple shell that allows to issue commands (followed by `<enter>`) that effect a running simulation.

A connection is made to a meeting point using telnet: `telnet host port`.

In telnet the following commands can be send to a simulation that is connected on the second port of the meeting point (if none is connected they are silently discarded):

'1' or 'n' or 'r+': do a single step

'n <steps>' : do `<steps>` steps, ie call steps() `<steps>` times

'b' : break/interrupt a simulation

'c' : continue an interrupted simulation

'i' : call the Felix-'init'-function

'r' : call the Felix-'init'-function and then 'step' continuously

'q' : quit the simulation

'B s v' : set a switch (button) 's' to value 'v' (0=FALSE; !0=TRUE)

'S s v' : set a slider 's' to value 'v' (v integer)

'O' : toggle output on/off

'D' : dump the switch and slider values to screen

'T v' : set the timer to 'v' (float in seconds)

Currently the timer is called after the execution of each step, ie., the timer and execution time of single steps add up to the total time between steps. This might change in the future. Similarly, the syntax above is not yet fixed. Note also that the telnet interface does not provide a prompt. Just type in commands linewise. You can leave telnet by pressing Ctrl-] and then 'q' at the telnet excape prompt. It is possible to reconnect to a running simulation.

8.2 Example: Remote Control

There is not much need for an example. Just some notes that re-iterate things already said:

- Only Felix programs compiled with the NO_GRAPHICS flag can be controlled remotely at the moment (see Makefiles in the Felix directories). These are programs compiled with no graphical user interface which is only the default for the parallel Felix implementation, pFelix.

- To tell a Felix program you want to control it remotely call it as `program host port`, where "program" is the program name and"host" and "port" are the hostname and listening port of the meeting point, respectively. host would be "localhost" if one connects through a tunnel otherwise the hostname of the machine the meeting point is running on.
- The meeting point must already be up, before the simulation starts
- You might also already want to be connected by telnet to the meeting point before you satrt the simulation. This, however, is not entirely necessary. You can connect and reconnect as often as you like.

8.3 Streaming Data

Note that the techniques described in this section are ver experimental. You can hang your simulations and perhaps even you computer.... BE WARNED

Many applications might need facilities to actually stream data in and out of the simulation per time step. At the moment it is only possible to write simulation data to disk, but not to receive it (more precisely, it *is* possible, but not recommended .... see the note at the beginning of this section).

To write(/send) data you have to open a socket that connects your Felix program to a server that is listening for incoming connections, can receive your data, and knows how to process it.

You probably would have to setup a second meeting point (on a different port) for the communication in order to get the data out of a computer cluster. If you don't need the remote simulation control described in the previous sections one meeting point would be enough.

Note that a simulation on a computer cluster can generate a huge amount of data in virtually no time. It is in general recommended to keep the communicated amount of data as low as possible. Think twice before you send anything ....

To connect a simulation to a remote application or a meeting point on the custer use:

```
sock = connect_tcp_client( hostname, port )
```

To write data to the socket use

`write_buffer( socket, buffer, size )` or

where socket is the socket returned by the call, buffer is a buffer to send, e.g. a vector or matrix, and size is the size of the object to send in bytes. A variant of the call allows to set additional flags that control some low-level options of the transmission

```
write_buffer( socket, buffer, size, flag )
```

Flags can be. e.g., MSG_MORE to tell the system there is more data to come and optimise transmitted package sizes, or MSG_DONTWAIT to tell it to send the data immediately. Several flags can be ORed together ... see "man socket" or "man send" if these man pages are installed on your machine. Otherwise search for introductions into socket programming on the internet.

In principle you can also read data using:

```
read_buffer( socket, buffer, size )
```

where `size` is the size of the data buffer `buffer` provided. However, you need to keep care of synchronisation between the simulation program and the program receiving the data, otherwise you might easily run into deadlock conditions.

8.4 Example 1: Data Streaming to a Disk on the Remote Machine

This example shows a simple receiver of data sent via a meeting point. It writes all incoming data into a file. It is not a Felix program but just links against the mylan.o-module of the Felix-kernel. The sender likewise does not need to be a Felix program.

1. Compile the code with something like `gcc -o rcvr rcvr.c mylan.o`. Of course the mylan header and object files need to be accessible.
2. Run the program with `rcvr host port file` where host and port specify a meeting waiting for connections, and file is the file to store arriveing data in.
3. Connect a sender to the secon port of the meeting point, e.g., telnet or a Felix program that opens a socket in main_init and writes data in step as outlined in the previous section.

```
# include <stdio.h>
# include "mylan.h"

int main(int argc, char *argv[])
{
  int res;
  char buffer[256];

  int sock = connect_tcp_client( argv[1], atoi(argv[2]) );
  FILE *fp = fopen(argv[3], "w");

  for(;;)
  {
    if ((res = recv(sock, buffer, 256, 0)) > 0 )
    {
      fwrite( buffer, 1, res, fp);
      fflush( fp );
    }
    else // error or connection closed
      break;
  }

  fclose(fp);
  close(sock);
}
```

Note, that if the sender terminates the above program will *not* also die, because it is connected to the meeting point and not directly to the sender. The meeting point doesn't report if the other port connects or disconnects, neither does it close a connection by itself (unless in error conditions). Thereceiver program therefore has to be killed explicitely with Ctl-C. The latter can lead to data loss if the file-buffer is not flushed. In the example we flush it after each write, but one could also setup a handler for the kill signal or redirect the system _exit routine in order to flush buffers on exit.

8.5 Example 2: Data Streaming to a Remote MIDI Device

This section shows an example that receives streamed data from a simulation and transforms them into sound events send to a MIDI port. It combines the mymidi and mylan modules. It has to be compiled as `gcc -g -o rcvmid rcvmid.c mylan.o mymidi.o -lasound`

The code below receives streamed data, ie., binary vectors of length 36 per step. A 1 on one of the 36 input lines means that a note is played. The lower 18 lines are mapped to one MIDI channel and the upper 18 to another one. Channels 5 and 9 are (usually) an electric piano and a percussion/drum set.

```
# include <stdio.h>
# include "mylan.h"
# include "mymidi.h"

int main(int argc, char *argv[])
{
  int res, i, sock, midiport;
  snd_seq_t *seq_handle;
  char buf[256], buf1[256];

  seq_handle = open_seq ( seq_handle, argv[0] );
  midiport = create_readable_port( seq_handle, argv[0], "out" );
  sock = connect_tcp_client( argv[1], atoi(argv[2]) );

  while (1)
  {
    if ((res = recv(sock, buf, sizeof(buf), 0))  > 0 ) // blocks!
    {
      for(i=0; i<36; i++)
      {
        int ch, note;

        if ( i < 36/2 ) {
          ch = 5; note = 40 + i;           // lower N/2 units
        } else {
          ch = 9; note = 40 + i - 36/2 ; }  // upper N/2 units

        if (buf[i] > buf1[i])      // note on
```

```
          send_noteon( seq_handle, midiport, ch, note, 127 );
        else if (buf1[i] > buf[i]) // note off
          send_noteoff( seq_handle, midiport, ch, note, 0 );

        buf1[i]=buf[i]; // save value for next step (on/off detection)
      }
    }
    else // error or connection closed
      break;
  }
  close(sock);
}
```

Setup is little tricky, because two meeting points are required, one for the control connection and one for the data stream. In addition if you want to make the generated MIDI events audible you have to connect the midiport to a synthesizer. Section ?? describes how to do that. The rcvmid program will just appear in the qjackctl tool as an additional readable port that can be connected to just any writable port that is available, e.g., qsynth.

1. Start qjackctl and qsynth
2. Start two meeting points on different ports, e.g., 12345 and 12346
3. Connect via telnet to the first meeting point
4. Start the rcvmidi program such that it connects to the second meeting point: `rcvmidid localhost 12346` (You have to use the proper host and port!)
5. Start the laninfo program described below such that it opens a control connection to the first meeting point `laninfio localost 12345`; the data streaming connection is made in the main_init routine of the program.
6. Connect the laninfo MIDI port to qsynth in the qjackctl tool
7. Set a proper time-step in telnet (T .1) and run the simulation (r)

I am aware that this is a pretty tedious procedure, but the code is currently just experimental; thinsg might get simpler in the future. Also, note that qjackctl, qsynth, and the meeting points need to be set up only once. However, each time the rcvmidi program is restarted it needs to be reconnected in qjackctl. The simulation program (here laninfo) automatically reconnects to the meeting points if it is restarted, but keep care of providing the proper hosts and ports: The control port needs to be connected to the telnet client, and the data port to rcvmid.

Here are code snippets how a simulation program would send data to the receiver program. It is assumed that the program uses a binary vector "spikes" of size N, which is send after computation a simulation step. In order to cooperate properly with rcvmid N must be 36. There should be a program laninfo somewhere in the example directories that implements an integrate and fire neuron network which streams spikes into the ethernet as shown below.

```
int main_init()
{
  ... init stuff ...

  sock = connect_tcp_client( "localhost", 12346 );
}

int step()
{
  ... do a simulation step ....

  write_buffer_1( sock, (char*)spikes, N, MSG_DONTWAIT );
}
```

Warning: If you try to write Felix programs with graphical user interface that receive data in their step-routine (or anywhere else) you can badly hang up the X11 server.

Chapter 9

Parallel Programming with Felix

NOTE: This chapter is quite preliminary

The present chapter describes recently developed parallel computing extensions to Felix. They are under development and many of them barely tested. Use at your own risk and don't expect too much!

Felix supports three types or parallelism: SSE-extensions, OpenMP for symmetric multi processors, and the message passing interface (MPI). The underlying concepts of these three technologies will be described in the subsequent chapters 9.2 to 9.4 individually. However, it is possible to combine all three frameworks in a single program. This makes sense in especially on computer clusters where each single node has several processor cores (see section 9.5). Such clusters will likely be the standard in future computer clusters.

Felix programs can be developed to run on serial or parallel target architectures. In general, at least some effort is necessary to parallelise a given serial code. However, it is at least in principle possible to write Felix programs that can be compiled and run on both, parallel and serial computers. Section 9.6 gives advise on how to write Felix programs of this kind.

9.1 History and Future

The very first Felix version was mainly intended to provide a graphical user interface for a parallel computer we had at the University of Ulm/Germany in the early 90th of the previous century (yes, I am almost a hundred years old!). This was a so-called "WaveTracer" comprising 4096 single bit processors running at an amazing 8MHz cycle-frequency. The processors could be arranged to form 1, 2, or 3-dimensional virtual arrays aiming primarily at simulations of wave equations and partial differential equations; the simulation of neural field models was possible as well. The programming made used of an ingenious C-dialect called "Multi-C", which I still believe was a brilliant development: It was C, enriched by a handful of parallel constructs for parallel data-types and data-transfer between nodes. Unfortunately the company WaveTracer died after a while and as it seems none of the other parallel hard- or software developers took over the conceptual ideas the WaveTracer system incorporated.

When single-CPU computers got faster than the WaveTracer, which happended surprisingly quickly, Felix was ported to standard architectures, first Sun-Workstations under Sun-OS and

early Solaris versions, later Linux PCs (and even later Cygwin ...).

More recently, computer clusters got cheap enough to become available for academic research. This caused Felix to be (back-)adapted to parallel environments again. The parallel Felix extensions therefore are very new, meaning that they are neither complete, nor very well developed, nor tested to a degree they should. So, be warned! In fact, they are under development and get extended as I find it useful for my research.

Felix supports three types of parallelism which intentionally should be freely combinable in applications. These technologies are abbreviated as SSE, OpenMP, and MPI – the first is a hardware technology for code-vectorisation, that latter two software-standards for the programming of symmetric multi-processor computers (SMP) and computer grids and clusters, respectively. None of them requires that you actually have a special parallel computer. You can install the necessary software on any Linux-box. This would allow you to develop parallel software on a Laptop or workstation, befor going big on a cluster. In fact, even better, every modern Intel or AMD processor supports SSE intrinsically, and the dual-core processor computers that currently start conquering the market have two physical processing units (SMP) per CPU-chip; they can naturally be programmed using OpenMP (or MPI) if full use of the two processor cores has to be made. Imagine that two cores per CPU are just the beginning: Intel has already presented its first 80-core wafer prototype and others will follow; 4 or 8 core CPUs will probably be available commercially in just a very few years.

9.2 SSE, BLAS, ATLAS

SSE is a hardware technology supported by each mordern AMD or Intel CPU. It was originally invented by Intel to speed up graphics and audio applications, ie., computer games, video, and all that kind of applications companies really can make money with.

SSE is indeed something like a co-processor in every single modern Intel or AMD CPU (I am not sure about MACs; but they switch to Intel CPUs as it seems). Each such processor has a main central processing unit which supports a certain instruction set and is most active during the execution of any standard program. Virtually all modern CPUs in addition have a math co-processor which can be used for speeding up computations of various mathematical functions like abs, sin, exp, and so on. Less well known is that since the Intel 386??? family or AMD ??? each processor has a further processing unit independent of the main arithmetic-logical-unit and math-co-processor that is useful for some kinds of parallel computations appearing often in graphics and audio processing. This hardware piece on modern chips is programmed by using the so-called SSE-extensions to the low-level assembler instruction set for that CPU.

The SSE standard basically provides a special register set on the CPU and accompanied assembler instructions which support some kind of math (but not a whole lot) supposed to be useful for graphics and audio applications. These register (by default 8 of them) are (at least on a 32 bit architecture) 128 bit wide, but the 128 bit can be divided into data-chunks of various size, ie., singned and unsigned integers of 8, 16, or 32 bit size, but also floating points of size 4 or 8 bytes (32 or 64 bits). Accordingly, these special units on any modern Intel or AMD CPU (yes, I am probably speaking about your computer) are able to process up to 16 8-bit integers, or 8 16-bit integers, or 4 32-bit floating points, or 2 64-bit floating points (doubles) at once. This supports a kind of "vectorisation", operations can be done in parallel on several numbers (ie., a "vector") at once. In principle every software could make use of this vectorisation, and indeed, commercial

compilers as well as newer versions of the gnu compilers are potentially able to compile code written in a higher programming language to make efficient use of the SSE extensions. (A full description of the SSE standard can be found in the respective documents available from Intels web-pages.)

Now, the "BLAS" is the so-called "Basic Linear Algebra Subroutines"-package which is available for Linux (MAC and Windows quite surely, too). It is a highly optimised package of linear algebra routines such as scalar, matrix-vector, and matrix-matrix multiplications. Some commercial products like Matab make use of the BLAS, which make their Matrix/Vector routines very efficient.

A standard Linux distribution does not usually have by default an optimised BLAS, because that library needs to be adapted to the precise target architecture. Most default Linux systems just have a default library (compiled for i368) that can be used by all pre-compiled programs on 99.9% of all PC architectures that need the library. However, you can update your BLAS to speed up such programs. Most of the improved BLAS versions do make use of the SSE extensions.

Two BLAS implementations are kind of standard at the moment: ATLAS- and Goto-BLAS.

ATLAS is an "automatically tuned linear algrebra system" that provides a BLAS and some routines on top of that (a subset of "LAPACK", a well-known "Linear Algebra Package" for solving linear equations, finding eigen-vectors, etc.). During compilation of ATLAS-BLAS, out of a large number (sometimes several hundreds and more) of possible implementations for a particular task like matrix-vector multiplication the best performing routines for the target architecture are chosen and put into the library. These top-performing routines can make use of the SSE CPU extensions and therefore the BLAS is mentioned under "parallel" Felix extensions.

GotoBLAS is a second BLAS implementation originally developed by Kazushige Goto. It is available (ie optimised) for a variety of target architectures and generally said to be the fastest BLAS implementation available. It does use hand-optimised (SSE-)assembler code.

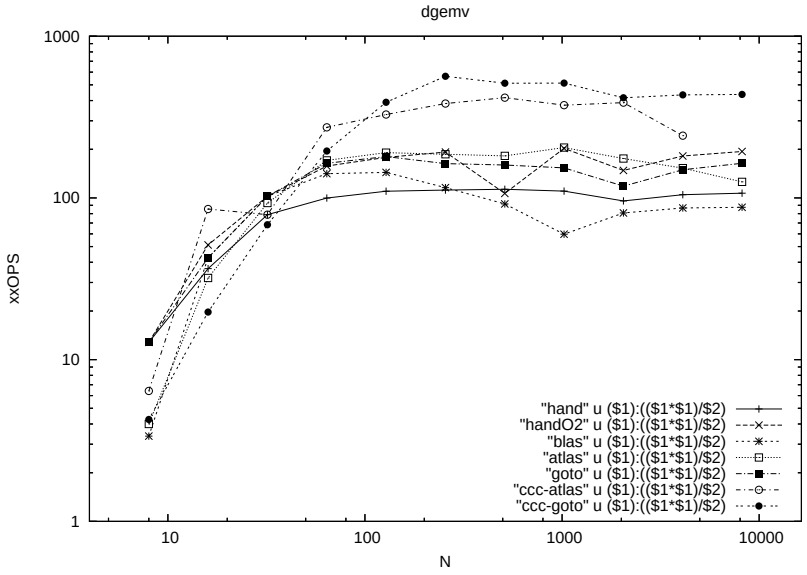

Figure 9.1: Typical performance for floating point *matrix-vector* multiplication on a Centrino laptop and a 4-core AMD opteron machine (CCC) of different raw and BLAS enhanced codes. x-axis indicates matrix/vcteor-size. See text for further explanations.

Figures 9.1 and 9.1 show the performance of matrix-vector and matrix-matrix multiplications for

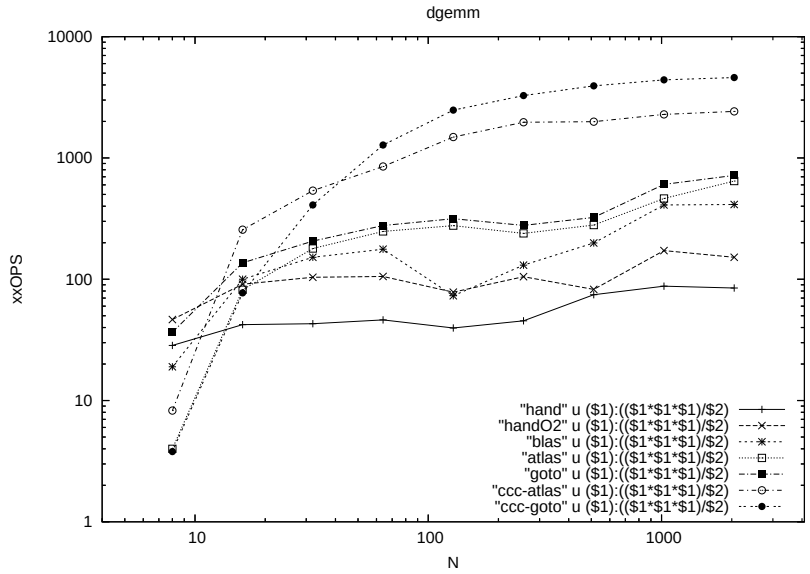

Figure 9.2: Typical performance for floating point *matrix-matrix* multiplication on a Centrino laptop and 4-core AMD opteron node (CCC) of different raw and BLAS enhanced codes. x-axis indicates matrix/vcteor-size.

floating point linear systems under different conditions. "ccc-goto" and "ccc–atlas" have been run on two processor dual core AMD 27?? nodes (4*2GHz; 4*1MB cache???); the other curves are for a single CPU Centrino Laptop (1.73GHz, 2MB cache). "hand" and "handO2 denote naive code (straight for-loops) either compiled without or with O2 optimisation using gcc 4.0.2. "blas" uses the default BLAS library on the Laptop, which performs worse than no optimisation at all in most of the studied range. "atlas" and "goto" indicate ATLAS and Goto-BLAS versions on the respective systems. Observe the quite impressive performance gain for optimised code, and that, of course, the numbers for the Centrino-Laptop and 4-core high-performance compute node are not directly comparable. Interestingly enough for small system size the single-CPU Laptop is faster than the 4-core AMD node.

Note: You don't need any BLAS library if you want to use Felix. It just can make some routines faster. At the moment the numbers of routines that potentially use BLAS-calls is actually more restricted than it could be. However, the floating point scalar-products, and matrix-vector products do use a BLAS library if this has been specified during compile time of the Felix libraries.

In order to let the Felix core use BLAS-routines whereever this is implemented to date it suffices to specify the -DWITH_BLAS flag during compile time. BLAS should not spawn threads (Goto-Blas can do this. It can be avoided using environment variables. See the repsective BLAS documents.)

9.3 OpenMP

Symmetric multi-processor computers (SMP) are systems that comprise a number or central processing units but share a common memory pool. Each processor can access the memory through a fast bus making memory access and data exchange potentially very fast.

Only since relatively recently SMP computers have been developed for the general market at

reasonable prices. Meanwhile, however, dual- and quad-processor computers are available at quite low prices and dual-core processors indicate a new trend that even aims at putting two (or more) processing units on the same chip. There are already many dual-core machines available, including Laptops. These all are SMP computers. Linux should support them automatically if you install an SMP-kernel.

OpenMP is an industry standard that supports programming SMP computers. It is not the most general approach for parallel programming (cf., e.g., concepts like Posix threads, PVM, or MPI), but for some kinds of applications it is *very* simple to use and can provide good speed ups. This includes neural network applications.

The most typical example for OpenMP-parallisation is "outer-loop"-parallelisation. It often is possible in numerical code where the same operations have to be performed on a large number of units. This is typically done in a big "outer" loop over the elements. OpenMP provides simple constructs to cut such loops into pieces of roughly the same size and distribute them over the availabe processors. In principle as single additional statement on top of an existing for-loop can be enough to parallelise it, e.g., a statement like

```
 for (i=0; i<N; i++)
   x[i] = func(i);
```

could result in

```
# pragma omp for private(i)
 for (i=0; i<N; i++)
   x[i] =  func(i);
```

This second version is automatically compiled into code distributed over the available processors. The variable i is declared private, because each process will need an independent copy of it. There are other constructs available for more general programming constructs than for-loops. There are also serious constraints that have to be taken into acount when parallelising code – for short, no two processes should ever try to potentially update the same variable at the same time (for that reason i has to be declared private in the pragma-statement; of course the function "func()" also is supposed to not assign values to variables possibly overlapping between processes. If this happens (a so-called "race-condition") the results of the computation are undefined. There are many cases, however, where assignments of variables are constraint to contiguous regions, e.g., in the range of a for-loop. In that case OpenMP-parallelisation is in general save to use. For further details, we have to refer the reader to the OpenMP specification, handbooks, and tutorials available in the Web.

A large number of functions in the Felix core automatically make use of OpenMP parallelisation if this is specified during compile time. It is also possible to use OpenMP macros in the user defined top-level init() and step() functions. It should, however, be avoided to call already parallelised Felix routines in parallelised regions in the top-level functions. Although the results would (probably) be well-defined (up to possible race-conditions), the code would spread an unnecessary large number of threads. Spreading threads for parallel computation always needs some overhead. It is therefore usually not advisable to parallelise very simple loops or to spawn more threads than processors are available.

9.4 MPI

MPI, the "Message Passing Interface", is an industry standard for communication between processes. These processes can run on the same or different computers, no matter where they are located (physical location only impacts the communication speed). Thus, MPI is useful for computer clusters and grids.

Simple MPI programs make use of a handful of statements only, although the full MPI standard defines over 120 different functions. These most simple commands just set up a logical network, and send and receive messages between nodes. For MPI-details in programming and usage we refer the reader to the many tutorials about MPI programming available on the Web. The following assumes basic knowledge about MPI programming.

Felix provides very simple constructs that don't do more than exchanging packages of various types of variables between processes.

The general philosophy is to run a number of copies of *the same program* on a number of available nodes (e.g., with `mpirun -np 3 programname` in the usual way to run MPI programs). Each copy has associated with it a number "myrank" that identifies it uniquely. Inside each running process code can therefore be executed conditionally depending on the rank of the process. After each simulation step, variables that are computed inside one process, but required in the next in another process have then to be communicated using MPI.

For that purpose every MPI-parallel Felix programm has to define a top-level routine "fmpi-connections()" that specifies which data has to be communicated. For each variable to be transmitted between two nodes a connection has to be setup using

```
void fmpi_connect( int node1, long var1,
                   int type, int size,
                   int node2, long var2 );
```

or the equivalent macro CONNECT (see example below).

"node1, var1" specify the source variable (typically an array of type CHAR, INT, or FLOAT, but can be a pointer to such an array, too, see below)

"node2, var2" is the target variable (must be an array, no POINTER type)

"size" is the number of elements in the array that have to be transmitted

"type" is the type of the data. The data basetype must be one of CHAR_TYPE, FLOAT_TYPE, or INT_TYPE. Note that Felix Vectors and Matrices are FLOAT_TYPE and bVectors CHAR_TYPE, such that it is admissable to specify the type as, e.g., bVECTOR or MATRIX. The basetype of the target variable must match that of the source. However, the source can in addition be a pointer type (similar as for display variables).

All connections have to be defined in a top-level function fmpi_connections(), e.g., like this:

```
void fmpi_connections()
{
  CONNECT( 0, var1, VECTOR, N, 1, var2 );
  CONNECT( 1, z1, POINTER TO bVECTOR, N, 2, z1 );
```

```
}
```

The first statement connects the float vector var1 of size N on processor 0 to var2 on processor 2. The second statement uses a POINTER variable, which gets dereferenced just befor transmission to a bVector of size N which is then transferred from processor 1 to variable z1 on processor 2. It is possible that source and destination are the same variables, but note that they will reside nonetheless on different machines.

Furthermore, if the same code is compiled using serial Felix, the CONNECT macro translates to empty code (but not the function, so use the macro!). Thereby, the code is discarded; nothing needs to be communicated if the program runs on a single processor. This supports writing code that can be compiled on serial *and* parallel machines without changing a singe line. Of course, using this feature needs a careful design of the code in order to have the serial and parallel codes consistent. There is typically at least a one-simulation-step delay introduced, because in the parallel versions communication occurs only after each simulation step, whereas in a serial program updated variables are immediately available.

9.5 Hybrid MPI/OpenMP Code

MPI and OpenMP can be combined in the same program.

The common free MPI versions (MPICH and LAM) are not threadsafe (most commercial implementations are). This means, if you use MPI within OpenMP parallelised regions the results are undefined.

Nonetheless, writing hybrid MPI/OpenMP-programs is possible, if care is taken of calling MPI-constructs only in OpenMP serial parts of the code. In that case only a single thread is doing the MPI-communication, which is safe.

Hybrid parallelism is possible in Felix. For that a number of MPI-processes are spawned that communicate as explained in section 9.4, but each of these processes in turn can spawn their own OpenMP threads. This is useful on SMP clusters with several CPUs per node. Communication between nodes can that way be done using MPI, but on the same node using threads and shared memory. Because communication via shared memory is usually faster than via a network this should result in speed benefits.

The following code is NOT Felix but just a simple C-example that demonstrates the principle.

```
// ompi.c -- simple test program for hybrid MPI/OpenMP paralellism

# include <stdio.h>
# include <omp.h>  // include OpenMP header
# include <mpi.h>  // include MPI header

# define NUMTHREADS 3  //  set number of OpenMP threads here

main( int argc, char *argv[] )
{
  int numtasks, rank;
```

```
  MPI_Init( &argc, &argv );
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks );
  MPI_Comm_rank(MPI_COMM_WORLD, &rank );

  omp_set_num_threads(NUMTHREADS);

# pragma omp parallel
{
  printf("MPI rank %d OMP thread %d\n", rank, omp_get_thread_num());
}

  MPI_Finalize();
}
```

The code needs to be compiled with an OpenMP-capable compiler (Intel, gcc 4.2 or higher) and linked against the proper MPI-libs (see also section A for further low-level info). It can then be run using, e.g., `mpirun -np 2 ompi` if "ompi" is the name of the executable. The number of MPI processes in the example would be 2 (specified by "-np 2" in the mpirun call), each of which spawns NUMTHREADS OpenMP threads. Each thread prints its MPI rank and thread number and exits.

Note that instead of setting the number of OpenMP threads explicitly one could also use the environment variable OMP_NUM_THREADS. This is quite usual and avoids having to recompile the code for different thread numbers. However, even if OMP_NUM_THREADS is set in your .bashrc, it is not necessarily exported to all target machines on all systems.

9.6 Parallelising Serial Felix Code

Serial code is compiled using the standard "Felix"-script, which links against libf (core routines) and libxf (XView extensions). For parallel code use the "pFelix"-script. This links against libpf.

Although libf and libxf are for serial code they can possibly make use of BLAS or OpenMP depending on how they have been compiled.

The parallel Felix lib "libpf" must be used for MPI and hybrid MPI/OpenMP.

9.6.1 OpenMP and *pflx*

To make life easier a couple of macros have been declared for writing parallelised code. If you use them you can even write programs that can be compiled and run with and without OpenMP.

```
# ifdef WITH_OMP
# define OMP_THREADS(_n) omp_set_num_threads(_n);
# define OMP_FOR(_x) ... // this shouldn't occur because preFelix removes OMP_FORs
# define OMP_ONLY(_x) _x
# else
# define OMP_THREADS(_n)
# define OMP_FOR(_x) for(_x)
```

```
# define OMP_ONLY(_x)
# endif
```

Observe that depending on whether the flag WITH_OMP is active during compile time (usually set in the Makefile) the macros expand into different code. If the Felix-script is used for compilation WITH_OMP will (usually) not be defined, but for the pFelix script it will.

Note that these settings only apply to your source code. Whether OpenMP is used in "libf", the Felix core library, depends on the value of OpenMP at compile time of the libraries, ie. in the Makefile in the Felix source directory. In the standard installation, libf would not contain OpenMP parallelised code.

There are several problems with the OMP_FOR macro: Actually this must have the form

```
OMP_FOR( <var> = <code> )
  < single statement or code-block enclosed by {}>
```

It should expand into

```
#pragma omp parallel for default(shared) private( <var> )
for( <var> = <code> )
  <single statement or code-block enclosed by {}>
```

The code segments not explicitely specified, of course, must translate into valid C-code.

The first problem now is that the `#pragma` phrase cannot be inserted by the preprocessor (at least I don't kow how to do it with macros). Instead a very simple preprocessor call "pflx" is used. This does nothing but searching a file for the string OMP_FOR and replacing the string in the sense above. "pflx" is called, when the Felix- or pFelix-scripts are executed. It generates a temporary file, which is then compiled into an executable.

The second problem with OMP_FOR is that the user has to make sure that the source-code does not contain assignments to memory locations which are potentially executed simultaneously in different threads. The values of such variables are undefined, but there will be no explicit warning or error message. Such variables in general need to be declared "private" in the reprective enclosing OpenMP-pragma or protected by other means (see OpenMP handbooks and tutorials). The only variable that is explicitly declared private if the OMP_FOR macro is used, is the run-index of the for-loop. This suffices in many situations I have experienced over the years. However, if you have code where some threads would potentially write/change the same shared memory locations, you can not use OMP_FOR, but have to use the original OpenMP pragmas.

Example: The following code is buggy

```
int i, j;
Matrix x;  // size nx * ny; allocated elsewhere
...
OMP_FOR(i=0; i<nx; i++)
  for (j=0; j<ny; j++)
    x[i*nx+j] =  ... something ... ;
```

The mistake is that j is a shared variable (by default). If several threads execute the outer for-loop, they all use the same copy of j (in shared memory), which they update asynchronously. This situation occurs often in simulations of two-dimensional field model. A simple cure is

```
int i;
Matrix x;  // size nx * ny; allocated elsewhere
...
OMP_FOR(i=0; i<nx; i++)
{
  int j;
  for (j=0; j<ny; j++)
    x[i*ny+j] =  ... something ... ;
}
```

Here the variable j is local to each thread and can only be changed by the respective thread. x is also a shared variable which gets values assigned, but note that the entries in that matrix are disjoint between threads, because different threads operate on different slices of the matrix.

9.6.2 MPI

A number of macros support writing MP-code.

```
# ifdef WITH_MPI
# define RANK(_x) if(myrank==(_x))
# define COND(_x) if(_x)
# define MPI_ONLY(_x) _x
# else
# define RANK(_x) // if(myrank==(_x))
# define COND(_x) // if(_x)
# define CONNECT(_x1,_x2,_x3,_x4,_x5,_x6)
# define MPI_ONLY(_x)
# endif

extern int myrank;
```

Observe that these macros expand to empty code when compiled serially (ie, if the compiler flag WITH_MPI is not set (usually in the Makefile, see **??**))

RANK and COND support conditional execution of code in conjunction with the global variable "myrank" which holds the unique MPI-rank of each process.

MPI_ONLY() can be used to enclose code that has to be executed only in an MPI environment (see example below.)

9.6.3 Example: Two interacting Neuron Pools

The code in this subsection simulates two pools of leaky-integrate-and-fire neurons, which interact mutually. It duplicates the variables and code from the previously used inf.c example program, but adds some code for the interaction and its control slider in the GUI.

The program is shown because it demonstrates how to write code using the macros explained in the previous section 9.6.2 that can either be compiled serially with GUI, but for parallel execution using MPI (or MPI/OpenMP) as well. The advantage would be that one can conveniently test a small version of the program with GUI, but run scaled-up large versions on a parallel computer without changing a single line of code. Both versions could even use the same environment files for parameter settings.

The idea is to cut the serial code into pieces that can be distributed across a number of MPI processes. The RANK() or COND()-macros are then used to select the respective code bits for execution in the individual processes. In order to set up the model properly one has to exchange data computed in one thread but needed in others, too. This is done by calls to the connect()-function in a top-level routine fmpi_connections(), see section 9.4.

The RANK, COND, and CONNECT-macros expand (basically) into empty code if the so prepared program is compiled serially. Using the macros appropriately, possibly in conjuction with the other macros in sections 9.4 and 9.3, can result in code that can be compiled serially and for parallel execution.

Here is one such magic codes (some parts have been cut out (mainly things related to display and output); the full source code should be in the Felix expl/parallel directory):

```
// infpairmpi.c

# include <felix.h>

# define N 100        /* number of neurons     */
# define tau 10.      /* membrane time const.  */
Vector  pot1, pot2;   /* potentials            */
Matrix  J1, J2;       /* connections           */
bVector o1, o2;       /* vector of spikes      */
Vector  v1, v2;       /* for help              */
int stp=0;

...

BEGIN_DISPLAY
....


BEGIN_OUTPUT
....


void fmpi_connections()
{
  CONNECT( 0, o1, bVECTOR, N, 1, o1 );
  CONNECT( 1, o2, bVECTOR, N, 0, o2 );
}


int main_init()
{
```

```
  randomize( time(NULL) + 100*myrank ); // not sure this safe ??????????????
  SET_STEPSIZE( .5 )

  RANK(0)
  {
    J1   = Get_Matrix( N, N );
    pot1 = Get_Vector( N );
    v1   = Get_Vector( N );
  }
  o1   = Get_bVector( N );

  RANK(1)
  {
    J2   = Get_Matrix( N, N );
    pot2 = Get_Vector( N );
    v2  = Get_Vector( N );
  }
  o2  = Get_bVector( N );
}

int init()
{
  int i;

  RANK(0)
  {
    Clear_bVector(N,o1);
    Clear_Vector(N,v1);
    for (i=0; i<N; i++)
      pot1[i] = equal_noise();   // random initialisation
    Make_Matrix( N, N, J1, 1./N , .4/N );
  }

  RANK(1)
  {
    Clear_bVector(N,o2);
    Clear_Vector(N,v2);
    for (i=0; i<N; i++)
      pot2[i] = 0;       // no random initialisation !
    Make_Matrix( N, N, J2, 1./N , .4/N );
  }

  stp=0;
}

int step()
{
  int i;

  RANK(0)
  {
```

```
    for (i=0;i<N;i++)
      leaky_integrate ( tau, pot1[i],
             0.01*( sinput + sJ*v1[i] + sJc*o2[i]
               + snoise*gauss_noise()
               )
                    );
    Fire_Reset( N,  pot1, 1.0, 0.0, o1 );
    bMult( N, N, J1, o1, v1 );
  }

  RANK(1)
  {
    for (i=0;i<N;i++)
      leaky_integrate ( tau, pot2[i],
             0.01*( sinput + sJ*v2[i] + sJc*o1[i]
               + snoise*gauss_noise()
               )
                    );
    Fire_Reset( N,  pot2, 1.0, 0.0, o2 );
    bMult( N, N, J2, o2, v2 );
  }

  stp++;

  MPI_ONLY(  // this ensures we don't run forever on the cluster
    if (stp >= 500)
    {
      MPI_Finalize();
      exit(0);
    }
  )
}
```

More explanations????????

The serial version of the code is compiled with "Felix infpairmpi" and run with "infpairmpi" from the command line as usual. The GUI should pop up as for standard serial Felix applications. If data storage is switched on, data of the first pool is written to file "pot1". Data of the second pool is not stored. The simulation runs until it is killed in the GUI.

The parallel version is compiled with "pFelix infpairmpi" and, e.g., run with "mpirun -np 2 infpairmpi" (It might be that you have to use other ways to run programs on your parallel computer, e.g., if the system adminstrator requires using a job scheduler). The parallel executable will not pop up a GUI. Data of the first pool will be written to "pot1-0" (by the first process); data of the second pool will not be saved, because no output files have been declared for the second process. The simulation exits after a certain number of steps (500).

Chapter 10

Example Programs

10.1 Leaky-Integrate-and-Fire Neural Network

```
/* Example-program: inf.c  */

# include <felix.h>

# define N   100    /* number of neurons      */
# define tau  10.   /* membrane time constant */

float I = 1.1,      /* Common input to units  */
      J0 = 1.1,     /* Coupling strength      */
      sigma = .1;   /* noise level            */

Vector  x;          /* potentials             */
Matrix  J;          /* connections            */
bVector z;          /* vector of spikes       */
Vector  v;          /* auxiliary variable     */


NO_DISPLAY

NO_OUTPUT


int main_init()
{
  /* init. random number generator and stepsize */
  randomize( time(NULL) );
  SET_STEPSIZE( .1 )

  /* allocate vectors and matrices */
  J = Get_Matrix( N, N );
  x = Get_Vector( N );
  z = Get_bVector( N );
  v = Get_Vector( N );
```

```
}

int init()
{
  int i;

  Clear_bVector(N,z);
  Clear_Vector(N,v);

  /* init. potentials with random values between 0 and 1 */
  for (i=0; i<N; i++)
    x[i] = equal_noise();

  /* init. J with gaussian distr. random numbers */
  Make_Matrix( N, N, J, 1.0/N, .4/N  );
}

int step()
{
  int i;

  for (i=0;i<N;i++)  // leaky integration for all neurons
    leaky_integrate ( tau, x[i],
                  I + J0*v[i] + sigma*gauss_noise() );

  Fire_Reset( N,  x, 1.0, 0.0, z );  // firing and reset

  bMult( N, N, J, z, v ); // redistribution of spikes
}
```

10.2 Coupled Chaotic Roessler Oscillators

Integrates differential equations with Runge-Kutta

Uses xy-plots

```
/*
 *  roessler.c -- coupled chaotic Roessler oscillators
 *                or  asymmetric damped harmonic oscillators
 */

#include "felix.h"


# define STEPSIZE .01
float t;

# define N 64       /* number of units     */
# define n  3       /* order of diff.system */
```

```
Vector  x;      /* x1 ... xN, y1 .... yN, z1 .... zN */
Vector  dxdt;     /* derivatives */
Vector domega;   /* used to give oscillators a gradient in properties */
Matrix  J;         /* connections  (if not meanfield couplings)  */
                   /* diffusive or random ....                   */

Vector  cfields;   /* coupling fields; either meanfield or diffusive
                      or random connectivity */
float xx1, yy1;

SwitchValue sosc = OFF;  /* Roessler or damped harmonic oscillators */
SwitchValue smean = ON;       /* mean field coupling */
SwitchValue sdiffusive = OFF; /* diffusive coupling */
SwitchValue swrand = OFF;    /* random connections */

SliderValue somega = 1000;
SliderValue sdelomega = 100;
SliderValue sepsilon = 100;
SliderValue sa = 150;


BEGIN_DISPLAY

 SWITCH( "osci type", sosc )
 SWITCH( "mean", smean )
 SWITCH( "diffusive", sdiffusive )
 SWITCH( "random", swrand )

 SLIDER( "mean omega", somega, 500, 1500)
 SLIDER( "delta omega", sdelomega, 0, 500)
 SLIDER( "coupling strength", sepsilon, 0, 500)
 SLIDER( "a", sa, 0, 500)

 WINDOW("signals")

  RASTER( "x", AR, AC, x, VECTOR,  N, 0, 0.0, 1.0, 2)

 WINDOW("MF-xy-plot")

  PLOT("x-y", AR, AC, &xx1, VECTOR, 1, 0, 0, 0, -20., 20.,
                      &yy1, VECTOR, 1, 0, 0, 0, -20., 20.,  2 );

 WINDOW("xy-plot")

  PLOT("x-y", AR, AC, x, VECTOR, N, n, 0, 0, -20., 20.,
                      x, VECTOR, N, n, 0, 1, -20., 20.,  2 );

 WINDOW("x(t)")

  GRAPH( "x1", AR, AC, x, VECTOR, N, 0, 0, 0, -20, 20 )
  GRAPH( "x2", AR, NC, x, VECTOR, N, 0, 1, 0, -20, 20 )
```

```
  GRAPH( "y1", NR, C0, &x[N], VECTOR, N, 0, 0, 0, -20, 20 )
  GRAPH( "y2", AR, NC, &x[N], VECTOR, N, 0, 1, 0, -20, 20 )

  GRAPH( "z1", NR, C0, &x[2*N], VECTOR, N, 0, 0, 0, 0., 20 )
  GRAPH( "z2", AR, NC, &x[2*N], VECTOR, N, 0, 1, 0, 0., 20 )

 WINDOW("MF")

  GRAPH( "x1", AR, AC, &xx1, VECTOR, 1, 0, 0, 0, -20., 20.)
  GRAPH( "x2", AR, NC, &yy1, VECTOR, 1, 0, 0, 0, -20., 20.)

END_DISPLAY

NO_OUTPUT


int main_init()
{

  SET_STEPSIZE( STEPSIZE )
  randomize( time(NULL) );

  J   = Get_Matrix(N,N);
  x   = Get_Vector(N*n);
  dxdt= Get_Vector(N*n);
  domega = Get_Vector(N);
  cfields = Get_Vector(N);
}


int init()
{
  int i;

  Clear_Vector(N, domega);
  Clear_Vector(N, cfields);
  Clear_Vector(N*n, x);
  Clear_Vector(N*n, dxdt);
  t = 0.0;

  for (i=0 ; i<N; i++)
  {
    domega[i] = -.5+(1.*i)/N;
    cfields[i] = 0.0;
    x[i] = 4.;
    x[N+i] = 4;
  }

  Clear_Matrix( N,N, J);
  Make_Matrix( N, N, J, 1, 1);
}
```

```
void derivs(x,y,dfdx)
float x;
float *y;
float *dfdx;
{
  int i,j;
  float omeg,a;

  a = .001*sa;

  for (i=0;i<N;i++)
  {
    if (sosc)  /* original roessler */
    {
      omeg = .001*(somega + sdelomega*domega[i]);
      dfdx[i]     = -omeg*y[N+i] - y[2*N+i] + cfields[i];
      dfdx[N+i]   = omeg*y[i] + a*y[N+i];
      dfdx[2*N+i] = .4+y[2*N+i]*(y[i]-8.5);
    }
    else   /* antisymm. undamped harm.osc. */
    {
      omeg = .001*(somega + sdelomega*domega[i]);
      dfdx[i]     =    a*y[i] -omeg*y[N+i] - y[2*N+i] + cfields[i];
      dfdx[N+i]   =  omeg*y[i] +   a*y[N+i];
      dfdx[2*N+i] = .4+y[2*N+i]*(y[i]-8.5);
    }
  }
}


int step()
{
  int i;
  float mf,epsfac;
  static float tlast=-1,phi1;


  rk4(x, dxdt, N*n, t, step_size, x, derivs);


  epsfac = .001*sepsilon;

  if(swrand)  /* different amplitude scaling in alternatives ... */
  {
    Mult( N, N, J, x, cfields );   /* good luck; first components of
                                      systems are first N vals of x */
    epsfac /= N;
  }
  else if(sdiffusive) /* open boundaries */
  {
    cfields[0] = x[1]-x[0];
```

```
    for(i=1;i<N-1;i++)
     cfields[i] = x[i+1]+x[i-1]-2*x[i];
    cfields[N-1] = x[N-2]-x[N-1];
  }
  else if (smean) /* mean field */
  {
    mf = Sum(N, x)/(float)N;
    for(i=0;i<N;i++)
     cfields[i] = mf;
  }
  else
  {
    for(i=0;i<N;i++)
     cfields[i] = 0.;
  }

  for(i=0;i<N;i++)   /* scale with coupling strength */
    cfields[i] *= epsfac;

  xx1 = Sum( N, x)/N;
  yy1 = Sum( N, &x[N])/N;

  t+=step_size;
}
```

10.3 Homogeneous Fields

```
/* ei-field.c -- two-dimensional excitatory/inhibitory neural field model
 *               probabilistic spiking neurons
 *               stimulus is a single long moving bar or two bars moving
 *               in parallel or antiparallel
 */


# include <felix.h>

# define tau1  3.
# define tau2  5.0

long stp = 0;
float sim_time, noise_fac;

Layer       input,
            pot1, pot2,
            f1, f2;

SpikeLayer  out1, out2;

# define L_SIZE11   8.0       /* FWHM in columns (float) */
# define M_SIZE11   8         /* Kernel dimension (int)  */
```

```
# define FM_SIZE11  (2*M_SIZE11+1)

# define L_SIZE12   8.0       /* FWHM in columns (float) */
# define M_SIZE12    4         /* Kernel dimension (int)  */
# define FM_SIZE12  (2*M_SIZE12+1)

# define L_SIZE21   8.0       /* FWHM in columns (float) */
# define M_SIZE21    4        /* Kernel dimension (int)  */
# define FM_SIZE21  (2*M_SIZE21+1)


UniKernel   kernel11,
            kernel12,
            kernel21;

Layer       link11,
            link12,
            link21;

# define barlength  25
# define barskip     0    /* 5  */
# define barsigma    7
# define BARINITOFFS 14.

double  yy1, yy2;
int bardirection = 1;


SwitchValue santi = OFF;
SwitchValue scent = OFF;

SliderValue sI1     = 85;
SliderValue sI2     = 85;
SliderValue sI     = 85;
SliderValue snoise = 20;
SliderValue sJ11  = 100;
SliderValue sJ12  = 40;
SliderValue sJ21  = 600;
SliderValue sspeed = 0;

BEGIN_DISPLAY

 SWITCH( "anti", santi)
 SWITCH( "center", scent )

 SLIDER( "Signal Input", sI, 0, 1000 )
 SLIDER( "E ", sI1, -200, 200 )
 SLIDER( "I ", sI2, -200, 200 )
 SLIDER( "noise", snoise, 0, 1000 )
 SLIDER( "J11", sJ11, 0, 500)
 SLIDER( "J12", sJ12, 0, 300)
 SLIDER( "J21", sJ21, 0, 1000)
```

```
 SLIDER( "speed", sspeed, 0, 1000);

 WINDOW("Excitation")

  IMAGE( " input ", AR, AC, input, LAYER, xsize, ysize, 0.0, 2.1, 1)
  IMAGE( "  pot1  ", AR, NC, pot1, LAYER, xsize, ysize, -.5, 1.0, 1)
  IMAGE( "  out1  ", NR, C0, out1, SPIKE_LAYER, xsize, ysize, 0.0, 1.0, 1)

WINDOW("Inhibition")

  IMAGE( " input ", AR, AC, input, LAYER, xsize, ysize, 0.0, 2.1, 1)
  IMAGE( "  pot2  ", AR, NC, pot2, LAYER, xsize, ysize, -.5, 1.0, 1)
  IMAGE( "  out2  ", NR, C0, out2, SPIKE_LAYER, xsize, ysize, 0.0, 1.0, 1)

WINDOW("Kernels")

  IMAGE( " k11", AR, AC, kernel11, CONSTANT LAYER,
                 FM_SIZE11, FM_SIZE11, 0.0, 1., 5)
  IMAGE( " k12", NR, AC, kernel12, CONSTANT LAYER,
                 FM_SIZE12, FM_SIZE12, 0.0, 1., 5)
  IMAGE( " k21", NR, AC, kernel21, CONSTANT LAYER,
                 FM_SIZE21, FM_SIZE21, 0.0, 1., 5)

END_DISPLAY


BEGIN_OUTPUT

 OUTFILE("phi1")

  SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON)
  SAVE_VARIABLE( "phi1 (pot1)", pot1, MATRIX, xsize, ysize,  SKIP | GRID ,
                  TSkip(2), Grid(26, xsize, 100, 32, ysize, 100) )

 OUTFILE("phi2")

  SET_SAVE_FILE_FLAG( THISFILE, ASCII, ON)
  SAVE_VARIABLE( "phi2 (pot2)", pot2, MATRIX, xsize, ysize,  SKIP | GRID ,
                  TSkip(2), Grid(38, xsize, 100, 32, ysize, 100) )

END_OUTPUT


static void init_bars(centerflag)
int centerflag;
{
  if (centerflag)  /* center */
  {
    yy1 = yy2 = ysize/2;
  }
  else
  {
```

```
    yy1 = BARINITOFFS;
    if (santi)
      yy2 = ysize-BARINITOFFS;
    else
      yy2 = BARINITOFFS;
    bardirection = 1;
  }
}

static void move_bars()
{
  if (scent)
  {
    init_bars(1);
    return;
  }

  if (yy1 > ysize-BARINITOFFS ||
      yy1 < BARINITOFFS)
    bardirection *= -1;

  yy1 += .001*bardirection*sspeed;
  if (santi)
    yy2 -= .001*bardirection*sspeed;
  else
    yy2 += .001*bardirection*sspeed;
}



static void smooth_bars( out )
Matrix out;
{
  int i, j, s1, s2, s3, s4;
  static double fac=0;
  double h;

  if (fac==0) fac = -.5/(float)(barsigma*barsigma);

  s2 = (xsize - barskip)/2;
  s1 = s2-barlength;
  s3 = (xsize + barskip)/2;
  s4 = s3 + barlength;

  for (j = 0; j<ysize; j++)
  {
    h = elem( out, j, s1, xsize) = triangle( fac * (yy1-j)*(yy1-j));
    for (i=s1+1; i<s2; i++)
      elem( out, j, i, xsize) = h;

    h = elem( out, j, s3, xsize) = triangle( fac * (yy2-j)*(yy2-j));
    for (i=s3+1; i<s4; i++)
```

```
      elem( out, j, i, xsize) = h;
  }
}



int main_init()
{
  int i;

  randomize(  time(NULL) );

  input = Get_Layer();
  pot1  = Get_Layer();
  f1    = Get_Layer();
  out1  = Get_SpikeLayer();
  pot2  = Get_Layer();
  f2    = Get_Layer();
  out2  = Get_SpikeLayer();

  link11 = Get_Layer();
  link12 = Get_Layer();
  link21 = Get_Layer();

  kernel11 = Get_UniKernel( FM_SIZE11, FM_SIZE11 );
  kernel12 = Get_UniKernel( FM_SIZE12, FM_SIZE12 );
  kernel21 = Get_UniKernel( FM_SIZE21, FM_SIZE21 );

  Set_Circ_Func_Uni_Kernel( kernel11, FM_SIZE11, FM_SIZE11, gaussian,
                            1., L_SIZE11, 0. );
  Set_Circ_Func_Uni_Kernel( kernel12, FM_SIZE12, FM_SIZE12, gaussian,
                            1., L_SIZE12, 0. );
  Set_Circ_Func_Uni_Kernel( kernel21, FM_SIZE21, FM_SIZE21, gaussian,
                            1., L_SIZE21, 0. );

  SET_STEPSIZE(0.5);
  noise_fac = sqrt(24.0/step_size);
}



int init( )
{
  int i,j;

  stp = 0;

  Clear_Layer(input);
  init_bars( scent );
  smooth_bars( input );

  Clear_Layer(pot1);
```

```
  Clear_SpikeLayer(out1);

  Clear_Layer(pot2);
  Clear_SpikeLayer(out2);
}

int step()
{
  int i,j,k;

  if (stp >= 36050)
    exit (0);

  /********************/
  /* compute stimulus */
  /********************/

  move_bars();
  smooth_bars( input );

  /********************/
  /* compute dynamics */
  /********************/

  /* excit. units */

  for (i=0; i<ysize; i++)
  {
    for (j=0; j<xsize; j++)
    {
      leaky_integrate( tau1, elem( pot1, i, j, xsize) ,
           0.001*(sI1 + sI*gauss_noise()*elem( input , i, j, xsize)
              + sJ11*elem( link11,i,j, xsize)
              - sJ12*elem( link12,i,j, xsize)
              + (snoise*noise_fac)*(equal_noise() - 0.5) ) ) ;
      elem( f1, i, j, xsize) = RAMP( elem( pot1, i, j, xsize) );
      elem( out1, i, j, xsize) = PROB_FIRE( elem( f1, i, j, xsize) );

    } /* END j */

    for (j=0; j<xsize; j++)
    {
      leaky_integrate( tau2, elem( pot2, i, j, xsize) ,
           0.001*( sI2 +  sJ21*elem( link21,i,j, xsize)
                 + (snoise*noise_fac)*(equal_noise() - 0.5) ) ) ;
      elem( f2, i, j, xsize) = RAMP( elem( pot2, i, j, xsize) );
      elem( out2, i, j, xsize) = PROB_FIRE( elem( f2, i, j, xsize) );
    } /* END j */
  } /* END i */


  bConvolute_2d_Uni( out1, kernel11, xsize, ysize, FM_SIZE11, FM_SIZE11, link11);
```

```
  bConvolute_2d_Uni( out1, kernel21, xsize, ysize, FM_SIZE21, FM_SIZE21, link21);
  bConvolute_2d_Uni( out2, kernel12, xsize, ysize, FM_SIZE12, FM_SIZE12, link12);

  stp++;

} /* END of step() */
```

Appendix A

Installation Guide

This appendix describes how to install the Felix simulation tool on serial and parallel computers. Lacking free time I never implemented proper autoconfiguration facilities. Therefore installation is quite low-level. However, a number of people have been able to install Felix on serial Linux boxes following the instructions below. Windows/Cygwin installations as well as installation of the parallel Felix extension can be a little more tricky.

The first part of this appendix describes the installation of the serial Felix version. This by default comprises the graphical user interface. Compiling Felix for parallelised code is described in the 2cd section. If you plan to use MPI, the GUI will not be available. The graphics works, however, with the SSE-BLAS and OpenMP code.

The following assumes that $FELIXDIR is the top-level directory of your Felix installation.

There should be a number of subdirectories (after unpacking)

$FELIXDIR/src : Source code of Felix kernel routines and libraries

$FELIXDIR/xview : Sorce code of X11 extensions used for the Felix-GUI

$FELIXDIR/lib : Felix libraries (created during compilation)

$FELIXDIR/expl : A number of example applications

$FELIXDIR/tools : A number of tools to transform Felix data files (e.g., for creating raster plots and gifs)

To compile the Felix core only the code in $FELIXDIR/src is needed. If you want the GUI you need in addition the code in $FELIXDIR/xview. These directories comprise several relevant Makefiles

$FELIXDIR/src/Makefile : main source code (compilation of serial lib libf)

$FELIXDIR/xview/Makefile : graphics extensions for X11 (compilation of serial lib libxf)

$FELIXDIR/Makefile : master Makefile to compile a serial application (envoked by the "Felix" command)

$FELIXDIR/src/Makefile.parallel : main source code (compilation of parallel lib libpf)

$FELIXDIR/Makefile.parallel : master Makefile to compile a parallel application (envoked by the "pFelix" command)

The first three Makefiles are required for compiling the serial libraries and code; the second two for parallel libs and code.

A.1 Standard (serial) Installation

A.1.1 Prerequisites

The Graphical user interface is built on X11 and a pretty old Widget tool called XView. XView is used for historical reasons. It was originally developed by Sun Microsystems who ceased supporting it in about 1995, when Motif became more dominant. It is still possible to get XView sources and binaries, but this gets more and more difficiult (in particular I don't know of any 64 bit packages).

Compilation of Felix presupposes an installed X11R6 package assumed to be in the standard location: /usr/X11R6 . X11R6 is by default contained in virtually all Linux installations. If this is the wrong path it has to be corrected in the Makefiles, i.e., those in ../src, ../xview and the top-level makefile.

The Felix GUI further requires installed XView libraries *libolgx* and *libxview*, e.g., in /user/openwin/lib

Felix further requires the XView development kit for include files, e.g., in /usr/openwin/include

It is possible to set an environment variable OPENWINHOME pointing at the location of the XView libs and include files during compilation.

Redhat/SuSe/Cygwin users: An XView rpm can be downloaded here http://www.physionet.org/physiotools/xview/

Ubuntu/Kubuntu/Debian users: The XView packages are in some (K)ubuntu repositories.

A.1.2 Serial Felix Installation

1. Create the Felix top-level directory ($FELIXDIR) where you want it.

   Default would be something like $HOME/felix.

2. Goto the target directory $FELIXDIR and unpack and untar sim.tar.gz **in** it by calling "tar -xzf sim.tar.gz"

3. Set environment variables for your shell. For the bash-shell (default in many Linuxes), add the following in $HOME/.bashrc :

```
export OPENWINHOME="/usr/openwin"
export FELIXDIR="\$HOME/felix"
export LD_LIBRARY_PATH="\$FELIXDIR/lib:/usr/X11R6/lib:\$LD_LIBRARY_PATH"
alias Felix="\$FELIXDIR/Felix"
```

The precise locations of the directories in the above exports possibly need to be adapted to your own file hierarchy. It might also be that /usr/X11R6/lib is already in your path or that the libs it contains are accessible by other means (in that case you can omitt it in the export above).

Beside that make sure "." (current directory) is in PATH (type echo $PATH in a shell and look for it). If it is not there you will have to type ./<progname> to run programs. Just <progname> would fail with "permission denied" or "program not found" or a similar error message.

4. Dont forget to execute "source .bashrc" in your running (bash-)shell after setting the environment variables. Alternatively, you can start a new shell so that the environment variables get set correctly.

5. Run "make install" in $FELIXDIR .

   If everything goes well this should compile the source code in $FELIXDIR/src and $FELIXDIR/xview, create the respective Felix core and GUI libraries, and move them to $FELIXDIR/lib.

   If this step is successfull you will have the (serial) Felix libraries *libxf* and *libf* in $FELIXDIR. Otherwise something went wrong.

6. Test a Felix example in $FELIXDIR/expl, e.g., inf.c :

   (a) change to the directory $FELIXDIR/expl

   (b) run "Felix inf" : the program "inf" should be compiled

   (c) run "inf" : "inf" should run and the graphical interface pop up

   If the test runs successful, you are ready to use the serial Felix version. Check out the examples in $FELIXDIR/expl .

A.1.3 Additional Notes

1. If you try to compile a felix program and get an error message that *panel.h, frame.h* or so are not found, then you don't have XView installed properly or haven't set the proper paths in the Makefiles.

A.2 Installation of Parallel Felix

The parallel Felix extensions are experimental code. Whereas much of the serial code (but not all) has been used for research for already many years, the parallel code is much more recent. I can't give much advise on it, it is in a pretty chaotic state, and it probably contains bugs .... feel free to improve it. Send patches or error warnings ...

Felix implements 3 levels of parallelism, which can at least intentionally be used simultaneously in any mix (this is mostly untested):

BLAS : Given proper BLAS/ATLAS libraries you might be able to use the SSE extensions of Intel and AMD CPUs. Note that you can use BLAS routines even if you do not have a multi processor system. BLAS routines support highly optimised Matrix/Vector Math. Some BLAS versions support automatic threading if you are on a multiprocessor SMP machine (e.g. gotoBLAS and, I believe, Intel MKL BLAS too). This, however, might interfere with level 2 OpenMP parallelism. If you are not careful, each OpenMP thread might spawn a number of BLAS threads. The BLAS libs usually support environment variables or other means to control the number of spawned threads.

OpenMP : OpenMP is a simple framework to parallelise outer loops on SMP multiprocessor machines. It automatically spawns threads that distribute separate parts of the loop over the available processors. Although simple to use OpenMP is suboptimal in various respects as compared to hand-coded threaded code. However, I have seen nice speed-ups for some of applications. gcc will support OpenMP from version 4.2 upward; the Intel compiler also implements the OpenMP standard. Since gcc isn't officially out yet, I use hacks to compile the OpenMP-parallel Felix code with icc, the Intel compiler. That makes some of the Makefiles look pretty nasty... (I have also compiled a pre-released gcc-4.2 snapshot. Seems to work, too.)

MPI : MPI is a message passing standard for multi processor systems including Symmetric Mulit Processors (SMPs) and Beowulf computer clusters. Felix uses very few very simple constructs to transport data between several co-operating processes in distributed Felix programs (see fmpi.c/h). In principle these are vectors/matrices transported between variables local to each process. Each process is running the same program but has a certain "rank" which can be used in the code to make parts of it selectively executable on some processes only. Check the paralle examples in $FELIXDIR/expl for more details.

The parallel version has its own Makefiles $FELIXDIR/src/Makefile.parallel and $FELIXDIR/Makefile.parallel which compile Felix versions without graphical interfaces. They contain flags for activating the different options.

You might also want to use these flags in the serial Makefiles. In that case you need to adapt the compiler settings and if you choose to activate MPI, you have to switch the graphical user interface off. BLAS and OpenMP parallelism, however, is comaptible with the GUI.

A.2.1 Prerequisites

You do not need a parallel computer to experiment with the parallel extensions. Each modern Intel or AMD CPU supports the SSE2 vectorisation which you may use in your BLAS version. You can also install and use MPI and OpenMP compilers/code on a serial machine. This way you can write and test code on, e.g., your laptop, before going on a bigger machine.

BLAS : A proper BLAS implementation, ie. ATLAS or gotoBLAS. The default BLAS that comes with many Linux versions is probably not speed-optimised (meaning that you can loose tremendous speed benefits for some matrix/matrix and matrix/vector operations. [At the moment BLAS is only used for some Felix functions — don't expect too much.]

OpenMP : As long as gcc 4.2 isn't available, you need another OpenMP capable compiler. There are some open source versions (I have used OmniMP, but wasn't happy with its optimisation

capabilities). I now use the Intel compiler, which has a free licence for single academic users. Thanks to Intel for that! You can also compile a prerelease of gcc 4.2 (or higher). This has the OpenMP standard built in. You need to adapt the Makefiles in that case.

MPI : The Felix MPI version works only without the graphical interface. It was developed for a computer cluster on which graphical interfaces make little sense. You can potentially compile with GUI in which case I would suspect each MPI process tries to open its own GUI. I never tested this.

You need gcc or icc or another C compiler and an MPI library. I use mostly MPICH(1) but at least previous parallel Felix versions worked also with LAM. I haven't checked MPICH(2) so far, but there is little reason why it should not work (one hears communication is considerably faster than MPICH(1)).

Note that Intel provides its own MPI libs, but I don't have them. Might be a useful investigation: Although I use icc, I link against the mpich libraries. That requires rather uncomfortable compiler settings (see Makefile.parallel).

One can run into problems with the MPI runtime environment not finding dynamic libraries. I therefore link part of the libs statically. That makes programs bigger. Alterantively, there are also linker switches to tell executables where to find the libs.

I use icc because to combine MPI with OpenMP one (obviously) needs an OpenMP capable compiler. Using Intel to date is the only (more or less) tested case (I have also testet a prerelease of gcc-4.2 very briefly; seems to work in principle). The Makefile.parallel is for icc, so have a look into it. You will see that I don't use the usual MPI compiler wrapper script, mpicc, but supply include and library directories etc directly to icc. You can probably avoid this, if you compile your own MPICH (or LAM?) using icc and use the mpicc version generated this way. I DO, however, use the "mpirun"-script of the MPICH standard installation.

A.2.2 Compilation of Parallel Felix

Compilation of parallel Felix follows the same steps as for the serial version. The instructions below compile a parallel library *libpf*, which can coexist with the serial libraries as compiled in the first section of this appendix (*libf* and *libxf*). You only have to use the script pFelix to compile a parallel application code against the right parallel libs.

To compile a parallel version of Felix without graphical user interface follow these instructions:

1. Beside the environment variables for the serial version you need to add another one for the parallel Felix script. In your .bashrc add

   alias pFelix="$FELIXDIR/pFelix"

2. Enable the desired flags in the parallel Makefile in the src and/or main directories:

   BLAS : Just enable -DWITH_BLAS in src/Makefile.parallel. [BLAS should work with and without graphical user interface, so that you could also use the serial Makefile if you want the GUI (doesn't work, though, if -DWITH_MPI is also set)].

   I did occassionally have some problems with linking against the right libs. You might have to adapt the Makefiles to get BLAS working

OpenMP : To use OpenMP switch -DWITH_OMP on in the Makefile and adapt it to use your OpenMP capable compiler (and linker, and archiver). The graphical interface should work with OpenMP, so that you can use the serial Makefiles, if you want graphical output.

MPI : To use MPI activate -DWITH_MPI and -DNO_GRAPHICS in the Makefile.

3. Adapt compiler, linker, archiver and flags, paths and libs in the Makefiles as necessary.
4. Delete any old object files present from compiling serial libs earlier by evoking "make clean" from a shell
5. Compile the parallel libs with "make -f Makefile.parallel par" in the src-directory.

   This should produce a library libpf.a in the lib-directory
6. You link against the parallel library libpf.a automatically if you use the "pFelix" script for compilation of your application code. This requires proper settings in the top-level Makefile.parallel.
7. Test an example from $FELIXDIR/expl/parallel, i.e., compile it using "pFelix prog" and run the generated executable using, e.g., *mpirun -np 2 prog*, where "prog" is the base program name (i.e., infmpi).

Note that serial programs and parallel code that uses MPI are not (in general) compatible. You need, e.g., to declare in the parallel code, which buffers are transported between processes. I will describe elsewhere how you can write applications that can be compiled parallel and serial (with GUI), and use even the same environment files.

A.2.3 Additional Notes

1. There is a compiler flag -DTIMING in the source Makefile. If this is switched on during compilation, timing information for the main parts of a Felix programm will be printed for each individual process.
2. If necessary, you can link Intel libs statically using -i-static flag of icc; this acts more specific than -static which links everything statically
3. You can tell a binary where to expect a library, e.g., mpiCC -Wl,-rpath=$INTEL/cc/9.0.030/lib/ -o mpitest mpitest.C

A.3 Windows / Cygwin

The serial Felix versions runs properly under Cygwin and the Windows operating system. It is considerably slower than under Linux, but still usable for, e.g., presentations.

An XView rpm can be downloaded here (together with instructions of how to install Cygwin and XView): http://www.physionet.org/physiotools/xview/

There is no obvious reason why the parallel Felix extensions should not work given the right tools (MPI, OpenMPI, BLAS). However, it has never be tried to compile parallel Felix on a Windows box.